DOCUMENT RESUME

ED 207 582 IR 009 700

AUTHOR Miller, Mark L.; Goldstein, Ira P.
TITLE SPADE: A Grammar Based Editor for Planning and Debugging Programs. AI Memo 386.
INSTITUTION Massachusetts Inst. of Tech., Cambridge. Artificial Intelligence Lab.
SPONS AGENCY Advanced Research Projects Agency (DOD), Washington, D.C.; National Science Foundation, Washington, D.C.
REPORT NO LOGO-33
PUB DATE Dec 76
GRANT NSF-EC40708X; ONR-N00014-75-C-0643
NOTE 70p.; For related documents, see IR 009 697 and IR 009 701-702.

EDRS PRICE MF01/PC03 Plus Postage.
DESCRIPTORS Classification; Computational Linguistics; *Computer Programs; Decision Making; Flow Charts; *Linguistic Theory; Planning; *Problem Solving; *Programming
IDENTIFIERS Debugging Aids; *Structured Planning and Debugging Editor

ABSTRACT

The Structured Planning and Debugging Editor (SPADE) is a new kind of interactive programming environment in which computer programs are generated by explicitly articulating planning decisions. The design of SPADE is based upon the development of a grammar of plans from a taxonomy of basic planning techniques. The utility of this approach to program definition is that a record of planning decisions, called the plan derivation, provides guidance for subsequent modification or debugging of the program. Moreover, this grammatical approach to planning allows the development of a taxonomy of bugs, as particular kinds of errors in applying the planning grammar. Following a linguistic analogy, five types of planning bugs are characterized: syntactic, semantic, pragmatic, circumlocutions, and slips of the tongue. The plan derivation can be accessed during subsequent debugging to aid in diagnosing the underlying cause of erroneous code. Repair is accomplished via replanning, in which a substructure of the derivation is replaced. The design of a debugging assistant for the SPADE environment called RAID is based on this theory. Thirty references are listed. (Author/LLS)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

ED207582

IR

Massachusetts Institute of Technology
Artificial Intelligence Laboratory

AI Memo 386 December 1976 Logo Memo 33

SPADE: A Grammar Based Editor For Planning And Debugging Programs

Mark L. Miller and Ira P. Goldstein

A grammar of plans is developed from a taxonomy of basic planning techniques. This grammar serves as the basis for the design of a new kind of interactive programming environment (SPADE), in which programs are generated by explicitly articulating planning decisions. The utility of this approach to program definition is that a record of these decisions, called the plan derivation, provides guidance for subsequent modification or debugging of the program.

Moreover, this grammatical approach to planning allows the development of a taxonomy of bugs, as particular kinds of errors in applying the planning grammar. Following a linguistic analogy, five types of planning bugs are characterized: syntactic, semantic, pragmatic, circumlocutions, and slips of the tongue. The plan derivation can be accessed during subsequent debugging, to aid in diagnosing the underlying cause of erroneous code. Repair is accomplished via replanning, in which a substructure of the derivation is replaced. A debugging assistant for the SPADE environment (RAID) is designed based on this theory.

The enterprise embodies Dijkstra's philosophy of programming in a structured fashion, but represents a more detailed study of planning and debugging techniques than has previously been attempted.

IR009700

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. It was supported in part by the National Science Foundation under grant C40708X, in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0643, and in part by the Division for Study and Research in Education, Massachusetts Institute of Technology.

Table of Contents

1. Introduction
    1.1. Background and Objectives
    1.2. Overview

2. A Grammatical Theory of Planning
    2.1. A Taxonomy Of Plans
    2.2. A Planning Grammar

3. The SPADE Editor
    3.1. SPADE-0: A Rudimentary Planning Assistant
    3.2. Towards SPADE-1, and Beyond

4. A Grammatical Theory Of Debugging
    4.1. Types Of Bugs
    4.2. Syntactic Planning Bugs
    4.3. Semantic Planning Bugs
    4.4. Pragmatic Planning Bugs
    4.5. "Circumlocutions" (Inefficiency Bugs)
    4.6. "Slips of the Tongue" (Execution Errors)

5. The RAID Debugging Assistant
    5.1. Diagnosis and Repair
    5.2. Aid In Diagnosing Syntactic Bugs
    5.3. Aid In Diagnosing Semantic Bugs
    5.4. Aid in Diagnosing Pragmatic Bugs
    5.5. Assistance in Repair

6. Conclusions
    6.1. Limitations and Extensions
    6.2. Further Applications

7. Notes
8. References

The authors would like to thank Carol Roberts for help with the illustrations.

1. Introduction

1.1. Background and Objectives

Our goals in this report are: (1) to understand the processes by which a programmer, whether human or machine, moves from a declarative statement of a problem to a procedural statement of its solution; and (2) to discover methods by which these processes can be facilitated. We see programming as involving two principle activities: *planning* and *debugging*. Most previous research has studied these two activities in an isolated fashion. This report presents a *unified* theory of planning and debugging, based on a linguistic analogy.

The investigation includes the design of an interactive programming environment called SPADE. *SPADE* is an acronym for *Structured Planning and Debugging Editor*. This name emphasizes two themes: (1) our perspective on programming as a process of planning and debugging; and (2) our expectation that SPADE-like systems will eventually help in achieving the structured programming movement's goals of program reliability, readability, extensibility, portability, and so on. The objectives for the SPADE programming environment are that it serve, not only as a practical application of the theory, but also as an experimental crucible for testing claims of the theory.[1]

In other papers the authors elaborate other dimensions of this linguistic approach to problem solving. [Miller & Goldstein 1976a] provides an overview of our research as a whole. [Goldstein & Miller 1976a] presents a long term research direction: applying the problem solving theory to the construction of a learning environment to teach elementary programming. In [Goldstein & Miller 1976b] the authors design PATN, an automated problem solver. In [Miller & Goldstein 1976b] the authors consider the use of grammars in the analysis of elementary programming protocols. In [Miller & Goldstein 1976d] the authors take steps toward automating this analysis task by designing a system called PAZATN.

1.2. Overview

The basis for SPADE's design is a unified problem solving theory which incorporates a fundamental linguistic analogy. The theory rests on a taxonomy of basic planning techniques. Planning, according to the theory, proceeds by a sequence of design decisions, in which the programmer selects a plan type and then carries out the subgoals defined by the application of that plan type to the current problem situation. This decision process is modeled by a context free grammar.

This analysis of planning leads to a taxonomy of program bugs as well.

Our claim that the theory *unifies* planning and debugging is based on the fact that classes of bugs are defined by tracing their origins to particular types of erroneous decisions in applying the planning grammar. Following a linguistic analogy, these planning bugs are characterized as: syntactic, semantic, pragmatic, circumlocutions, and slips of the tongue.

The SPADE system will provide an interpreter for context free grammar rules. It will provide bookkeeping facilities, maintaining a record of the planning decisions made in the application of each rule. This data structure generated by the grammar is called the *plan derivation*. Programs are merely the terminal strings of such derivations. Hence, SPADE should encourage programmers to *articulate* their planning decisions, rather than merely leaving the plan implicit in the resulting code.

The derivation structure created during planning episodes can be accessed during subsequent debugging episodes to aid in diagnosing the underlying cause of malfunctioning code. Repair would then proceed via replanning, in which a substructure of the plan derivation is replaced. One result of this repair would be that the purely hierarchical derivation *tree* is replaced by a *chart* of alternative derivation trees. Diagnosis and repair techniques based on this theory are to be implemented in a debugging assistant called RAID (for RAtional Implementation of Debugging). RAID will be a component of the SPADE environment.

This paper presents the design for SPADE. We plan to implement the system. The implemented system will serve as the basis for a set of experiments exploring aspects of the theory, such as the relative effectiveness of alternative planning grammars. Examination of session transcripts coupled with systematic interviews of SPADE users will provide evidence for answering the following sorts of questions:

1. Do users find the planning grammars adequate; or are there planning decisions which simply cannot be made given the restrictions of the grammar?

2. How much of the grammar would remain the same in moving from one application to another? We initially plan to implement the domain dependent portion of the grammar for the Logo elementary graphics programming domain.[2] Later we intend experimenting with planning grammars for different domains, to include: the "blocks world," a set theory world, and an elementary calculator world.[3]

3. Do the plan derivation structures generated by the grammar serve as useful documentation, aiding one programmer in understanding and modifying programs written by another?

4. How effective is the system as a pedagogical alternative for teaching programming and problem solving? Can its effectiveness be attributed to such factors as greater articulation of planning and debugging strategies?[4]

The answers to these questions, in turn, will shed light upon a larger question addressed by the enterprise: does computational linguistics provide a valuable set of formal concepts and algorithms for constructing a theory of problem solving?

Section two presents our theory of planning. The third section introduces the SPADE system. Our theory of debugging, and its embodiment in RAID, are the topics of sections four and five. We conclude by discussing limitations, extensions, and applications to structured programming, automatic programming, and protocol analysis.

2. A Grammatical Theory of Planning

> It would help a great deal if we had a general language specially designed for talking about Plans.... Such a language would, presumably, give us a convenient notation for such aspects as flexibility of Plans, the substitution of subplans, conditional and preparatory subplans, etc. For example, it does not particularly matter in what order Mrs. Jones chooses to run her errands when she gets to town. The ... subplans can be permuted in order, and so we say that this part of her Plan is flexible. But she cannot permute the order of these with the subplan for driving to town, or for driving home. That part of the plan is inflexible. Some subplans are executed solely for the purpose of creating the conditions under which another subplan is relevant. Such preparatory or mobilizing subplans cannot be freely moved about with respect to the other subplans that they anticipate. Another important dimension of freedom that should be analyzed is the interchangeability of subplans. Mrs. Jones can drive to town over a variety of equivalent routes. The variety is limited only by the condition that they terminate when one of her three alternative destinations is reached, since only then would the next part of her Plan become relevant. Given a satisfactory Plan and a statement of the flexibility and substitutability of its subplans, we should then be able to generate many alternative Plans that are also satisfactory. And we should like to have ways for deciding which combinations of Plans are most efficient....
>
> [Miller et al. 1960]

2.1. A Taxonomy of Plans

To arrive at a syntax of plans, we begin by formulating a taxonomy of planning methods. Figure 1 presents a taxonomy of a variety of common planning techniques.[5] We arrived at this taxonomy partly by introspection, partly by examining problem solving protocols [Miller & Goldstein 1976b], and partly by studying the analyses of problem solving provided by Polya [1957, 1962, 1965, 1967, 1968]. The taxonomy is incomplete: different domains would emphasize different planning techniques. Yet there is certainly a core set of planning techniques common to all domains.[6]

The initial division in the taxonomy is into planning by identification, by decomposition and by reformulation. The first category captures those methods which solve the problem by identifying it as one which is already known. The second provides guidelines for breaking the problem into pieces. The third

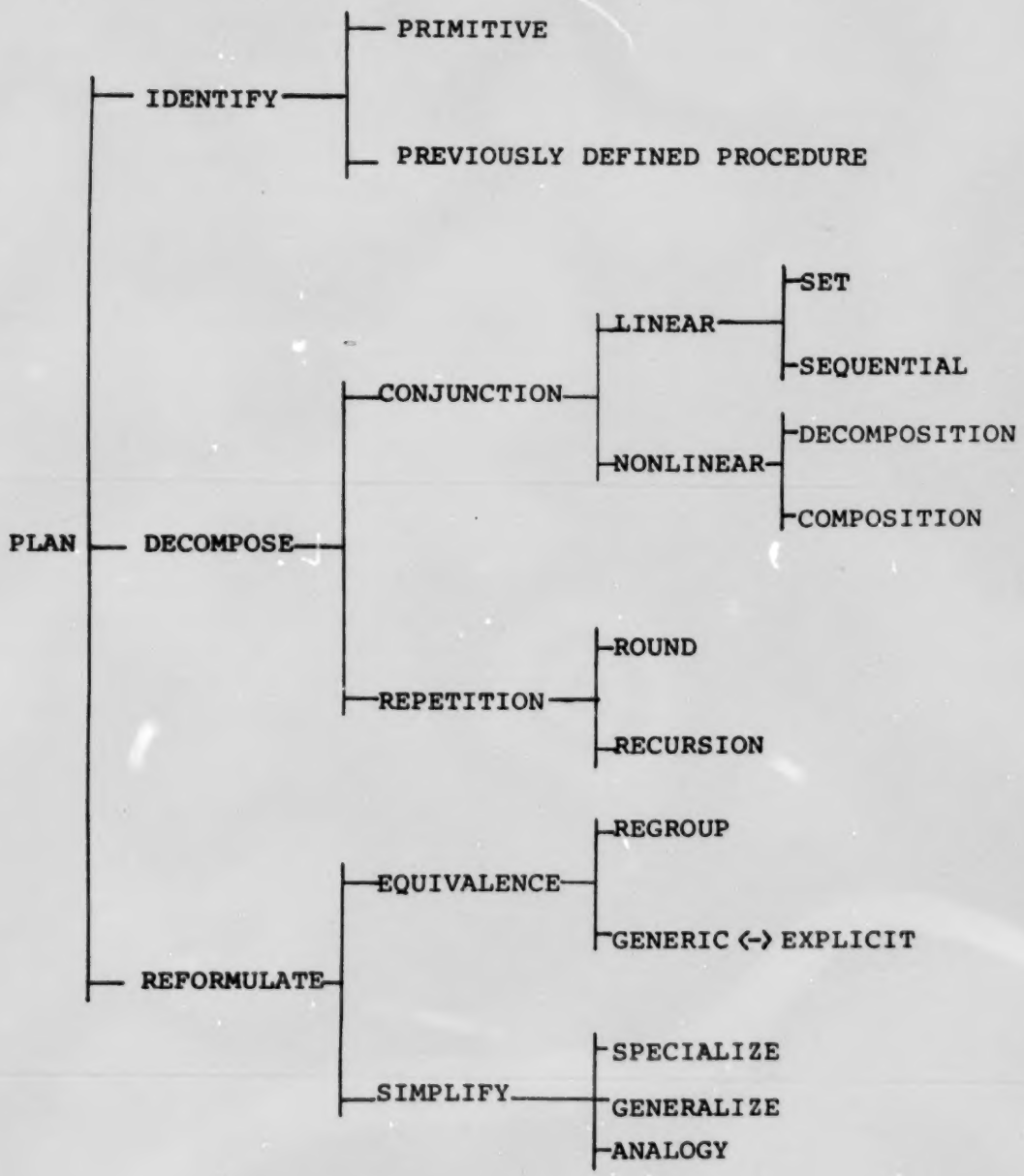


FIGURE 1
TAXONOMY OF PLANNING CONCEPTS

includes techniques that attempt to reformulate the problem into a form more amenable to identification or decomposition.

For any domain, there are primitives and previously solved problems. Hence, the identification class breaks into these two sub-categories. Of course, there can be enormous subtlety in how a problem is recognized as an instance of a previously solved case. Constructing a taxonomy does not resolve this issue. In [Goldstein & Miller 1976b] we introduce formal descriptions of the problem domain, and hence can address this issue more precisely.

There are many decomposition techniques. The taxonomy of figure 1 cites only two: decomposition into conjunctive subgoals and decomposition into a single subgoal, repeated some number of times. Other decomposition techniques are appropriate for problems that can be decomposed into a disjunctive set of subgoals, or into a negation of some goal. Conjunction involves the critical question of whether each conjunct can be solved independently of the others, or whether there are interactions. Repetition divides into solution by simple iteration of a single subgoal or solution by full recursion.

Reformulation is perhaps the subtlest of the planning categories. It includes finding an equivalent formulation of the problem which presumably is easier to solve or a critical simplification whose solution is a stepping stone to the solution of the original problem. Occasionally, one may even reformulate a problem into a stronger form: such as constructing an example when only an existence proof is required.

How can we further explore this set of planning concepts? Our first step is to be more explicit about the decision process involved in selecting planning methods from this taxonomy.

2.2. A Planning Grammar

We view planning as a process in which the problem solver selects the appropriate plan type and then carries out the subgoals defined by that plan applied to the current problem.[7] From this viewpoint, the planning taxonomy represents a decision tree of alternative plans. This decision process can be formalized by a context free grammar.[8] A grammar is chosen to present these rules because it provides a simple and compact representation, useful for characterizing the hierarchical structure of planning. We would not argue that a context free grammar is the appropriate formalism for representing a complete theory of problem solving -- elsewhere we employ a more elaborate formalism. However, we believe that the grammar represents a useful abstraction of the decision points in the planning process.

The top level rule in the problem solving grammar is:

P1: SOLVE -> PLAN + [DEBUG][9]

The nonterminal SOLVE is formally analogous to the nonterminal SENTENCE in a linguistic grammar for parsing or generating sentences. P1 states that planning is first used to generate a plan, with subsequent debugging then being required to complete the solution. Of course, the plan may be entirely correct. For this reason, DEBUG is in brackets, indicating that it is an optional constituent. We shall have more to say about debugging in a later section.

The planning taxonomy characterizes the planning process as involving three mutually exclusive plan categories: identification, decomposition, and reformulation. Hence, in planning, the problem solver must choose among these alternatives. We represent this by the disjunctive rule P2.

P2: PLAN -> IDENTIFY | DECOMPOSE | REFORMULATE

Now let us consider the details of each of these planning categories. Identification consisted of using a primitive or using a previously solved problem. This is described by P3.

P3: IDENTIFY -> PRIMITIVE | DEFINED

The first alternative leads to the use of primitives from the particular problem domain being investigated.

The planning theory is modular, and independent of the application domain. But it is obviously critical to illustrate its applicability by concrete examples. In this report, we use the Logo elementary graphics programming domain as our source of examples. The task in this domain is to draw pictures with a cursor called the "turtle" by means of programs that move the cursor on the screen. Figure 2 illustrates the grammar rules for the primitives of this domain. Figure 3 illustrates a typical goal undertaken by beginning programmers, a "wishingwell picture."

The second identification alternative, DEFINED, involves retrieving a solution from the library of previously defined solutions and inserting it into the current solution. These two steps are captured by the rule P4.

P4: DEFINED -> USE-CODE & GET-FILE[10]

We now turn to the second major planning category, decomposition. Two important decomposition techniques are conjunctive plans, in which the problem is sub-divided into independent parts, and repetition plans, in which the problem is

Figure 2. Grammar Rules for Logo Primitives

L1.	PRIMITIVE	->	VECTOR | ROTATION | PENSTATE
L2.	VECTOR	->	FORWARD|BACK + "number"
L3.	ROTATION	->	LEFT|RIGHT + "number"
L4.	PENSTATE	->	PENUP | PENDOWN

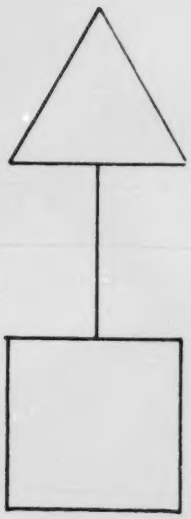


FIGURE 3
WISHINGWELL PICTURE

characterized in terms of a sub-problem repeated some number of times.

P5: DECOMPOSE -> CONJUNCTION | REPETITION

Were we to include other plans for decomposing problems, such as disjunctive plans, this rule would be extended by adding additional options.

The taxonomy shows conjunction as splitting into two cases: linear and nonlinear. The linear case is intended to represent the situation wherein the conjuncts can be solved entirely independently. The solution to the original problem then becomes simply sequencing the solutions to the subgoals; or, in some cases, executing them in any order, i.e the independence extends even to the composition process. Solving for the roots of a factored polynomial is linear (each root can be solved for independently) and the composition is set structured (the order of the solution does not matter). Solving for the sub-pictures of the wishingwell shown earlier is independent, but to obtain the desired relations between the parts, some specific sequence must be established. Rule P6 defines the two cases for conjunction:

P6: CONJUNCTION -> LINEAR | NONLINEAR

Rule P7 specifies the two alternatives for a linear solution:

P7: LINEAR -> SET | SEQ

P7 is incomplete: The composition of independently solved subgoals might be in parallel, or via some interrupt control structure. A goal of our research is to develop the depth and breadth of the taxonomy and its associated procedural forms so as to include such constructs.

A sequential plan consists of a sequence of actions, each consisting of a main step followed by an optional interface; these are preceded and followed by optional setup and cleanup steps.

P8: SEQ -> [SETUP] + <MAINSTEP + [INTERFACE]>* + [CLEANUP]

The essence of a sequential plan is that the solutions to the main steps can be designed independently of each other.

A set plan is simpler: the independence of the composition implies that no setup or cleanup steps are necessary.

P9: SET -> <STEP>*

For the programming domain, a setup, main step, interface, or cleanup

consists of either the addition of a line of code or a recursive application of SOLVE.

```
P10: SETUP          -> STEP
P11: MAINSTEP       -> STEP
P12: INTERFACE      -> STEP
P13: CLEANUP        -> STEP
P14: STEP           -> ADD | SOLVE
```

The grammar now admits potentially infinite recursion. What is not formalized by the context free grammar is the fact that SOLVE is always attempted with respect to some specific problem and in a definite context. Successful planning involves solving successively simpler problems until a direct solution in terms of the answer library is possible. The semantic and pragmatic components, formalized in [Goldstein & Miller 1976b], would constrain the potentially infinite recursion allowed by the grammar.

Similarly, the grammar does not capture the distinction between a setup, main step, and cleanup: they are all simply steps. There is, however, a semantic distinction. For example, the distinction between a main step and a setup depends on whether the code is designed to directly accomplish some subgoal -- a main step; or to establish some prerequisite for accomplishing some subgoal -- a setup. For example, in the Logo graphics domain, main steps generally involve drawing a visible part of the picture while setup steps have the goal of invisibly modifying the position or heading of the turtle between adjacent main steps. The Mycroft program [Goldstein 1974] included a program annotator that made such distinctions by comparing the picture drawn by the code with a predicate logic description of the intended picture.[11]

P15 states that repetition plans can be accomplished either by simple loops or by full recursion. (The latter is not elaborated here.)

```
P15: REPETITION     -> ROUND | RECURSION
```

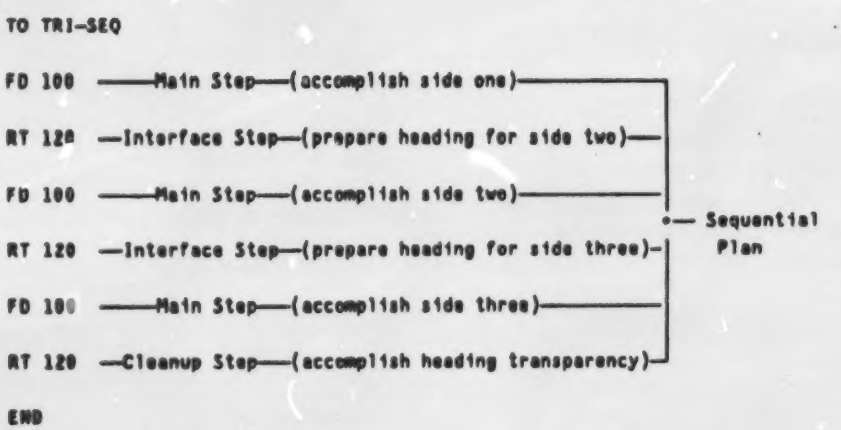
A round plan is the simple looping case, which can be accomplished either by iteration or by tail-recursion. (Tail-recursion is the restricted case wherein the recursion is constrained to be the last line of the program. It is computationally equivalent to a simple loop structure.) The following rule captures this:

```
P16: ROUND          -> ITER-PLAN | TAIL-RECUR
```

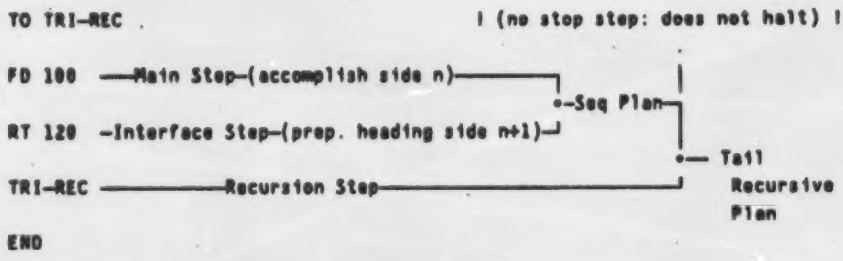
Figure 4 illustrates a triangle being accomplished by three different Logo programs. These correspond to the use of a sequential plan, a recursive round plan and an iterative round plan. The annotations in parentheses, stating

Figure 4. Accomplishing A Triangle

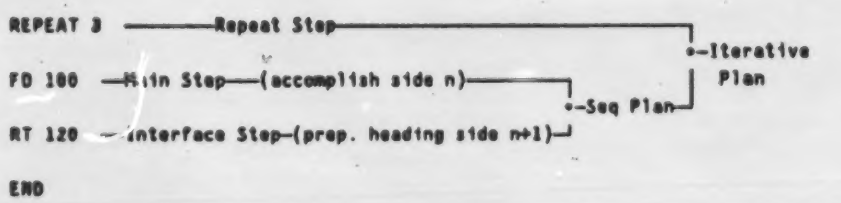
(Sequential Plan)


TO TRI-SEQ
FD 100 ——Main Step——(accomplish side one)
RT 120 —Interface Step—(prepare heading for side two)—
FD 100 ——Main Step——(accomplish side two)
RT 120 —Interface Step—(prepare heading for side three)—
FD 100 ——Main Step——(accomplish side three)
RT 120 —Cleanup Step——(accomplish heading transparency)
END
Sequential Plan

(Tail Recursive Plan)


TO TRI-REC .
! (no stop step: does not halt) !
FD 100 —Main Step—(accomplish side n)
Seq Plan
RT 120 —Interface Step—(prep. heading side n+1)—
TRI-REC ——Recursion Step
Tail Recursive Plan
END

(Iterative Plan)



what the planning step is intended to accomplish, are semantic descriptions not generated by the grammar. The grammar must be supplemented by semantic interpretation rules to allow for such analysis.

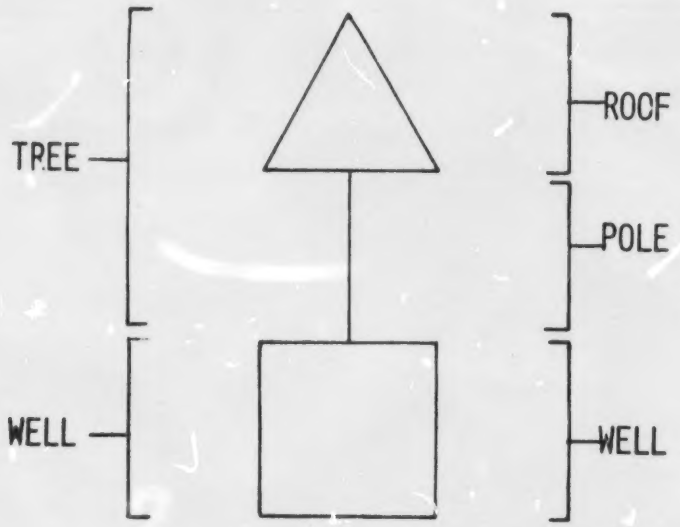
Tail recursion may be represented as a sequential plan plus recursion and stop steps. Iteration is similar.

```
P17: ITER-PLAN      -> "repeat step" + SEQ
P18: TAIL-REC       -> STOP-STEP + SEQ + REC-STEP
P19: REC-STEP       -> "recursive program call"
P20: STOP-STEP      -> "stop program call"
```

Reformulation, the third major planning category, should be briefly mentioned. Figure 5 provides a simple example of reformulation by regrouping the parts: a wishingwell, originally decomposed into a roof, a pole and a well, is later viewed as decomposable into a tree and a well. Reformulation techniques depend intimately on the problem description. Hence, we do not consider them further in this report. The subset of the planning grammar employed here is summarized in figure 6.

Figure 5

REFORMULATING THE WISHINGWELL IN TERMS OF A TREE



SINCE SPADE-0 HAS NO PROBLEM DESCRIPTION, IT MAY NOT ALWAYS BE APPARENT WHEN A REFORMULATION HAS OCCURRED. SOMETIMES IT WILL BE APPARENT, THOUGH, FROM THE DIALOGUE. FOR EXAMPLE:

1A. WHAT ARE YOUR SUBGOALS?
1B. ROOF, POLE, WELL.

2A. WHAT WOULD YOU LIKE TO DO?
2B. REDO CHOICE 1.

3A. CHOICE 1 UNDONE.
WHAT ARE YOUR SUBGOALS?
3B. TREE, WELL.

4A. RULE FOR TREE IS: SOLVE →…

Figure 6. G2: A Grammar of Plans

```
P1:  SOLVE        -> PLAN + [DEBUG]
P2:  PLAN         -> IDENTIFY | DECOMPOSE | REFORMULATE
P3:  IDENTIFY     -> PRIMITIVE | DEFINED
P4:  DEFINED      -> USE-CODE & GET-FILE
P5:  DECOMPOSE    -> CONJUNCTION | REPETITION
P6:  CONJUNCTION -> LINEAR | NONLINEAR
P7:  LINEAR       -> SET | SEQ
P8:  SEQ          -> [SETUP] + <MAINSTEP + [INTERFACE]>* + [CLEANUP]
P9:  SET          -> <STEP>*
P10: SETUP        -> STEP
P11: MAINSTEP     -> STEP
P12: INTERFACE    -> STEP
P13: CLEANUP      -> STEP
P14: STEP         -> ADD | SOLVE
P15: REPETITION   -> ROUND | RECURSION
P16: ROUND        -> ITER-PLAN | TAIL-RECUR
P17: ITER-PLAN    -> "repeat step" + SEQ
P18: TAIL-RECUR   -> STOP-STEP + SEQ      + REC-STEP
P19: REC-STEP     -> "recursive program call"
P20: STOP-STEP    -> "stop program call"
```

3. The SPADE Editor

How can we validate a particular grammar? How can we judge whether the grammar captures at some level of abstraction the set of planning decisions involved in solving problems for some domain? One traditional methodology for AI is to develop an automated problem solving system. The grammar, however, is insufficient for this. Semantics and pragmatics are required to make our theory deterministic. (We develop this in [Goldstein & Miller 1976b].)

But another methodology is possible. This involves incorporating the grammar into an intelligent editing system to augment the capabilities of the human problem solver. The critical question is whether such an intelligent support system successfully aids the user. In this section we design SPADE, an editor for defining programs that incorporates our planning grammar.

3.1. SPADE-0: A Rudimentary Planning Assistant

The name *Structured Planning and Debugging Editor* emphasizes the link between the problem solving theory being evolved here and the structured programming movement. Dahl, Dijkstra, and Hoare [1972] properly argue for programs that reflect coherently structured problem solving. But they do not develop a theory of planning in any great detail. Our effort in this direction, therefore, naturally supplements the examination of programming style initiated by Dijkstra and colleagues.

Figures 7a and b illustrate an interaction with SPADE-0, our proposed grammar based editor.[12] The user is interested in defining a Logo program for drawing the wishingwell shown earlier. In the SPADE environment, this is done by applying the planning grammar in generative mode. The user begins by specifying whether the plan is to be an identification, decomposition or reformulation. The editor keeps track of the pending subgoals, allowing the user to move from one goal to another.

Consider lines 6a and b of the SPADE-0 scenario.

6a. Do you wish to include the optional constituent CLEANUP?
6b. >LATER

These lines illustrate the user suspending one subgoal, without completing it, in order to pursue another. Although SPADE's default is to suggest subgoals in "top down" order, the system will offer the user the freedom to deviate from this default, pursuing the set of required goals in any other order. The decision as to whether a CLEANUP step is needed in the sequential plan is part of the

Figure 7a. SPADE-0 Scenario (part a)

The preliminary version of SPADE is little more than a bookkeeper for the user's subgoals. It is currently being implemented by assigning an interpretive procedure to each grammatical primitive.

1a. What is the name of your top level procedure?
1b. >WW

2a. Rule for WW is: SOLVE -> PLAN + [DEBUG].
Rule for WW-1 is: PLAN -> IDENTIFY | DECOMPOSE | REFORMULATE.
What now?
2b. >DECOMPOSE

.
.
.

3a. Rule for WW-4 is: LINEAR -> SEQUENTIAL | SET.
What now?
3b. >SEQUENTIAL

4a. Rule for WW-5 is: SEQ -> [SETUP] + <MAIN + [INTERFACE]>* + [CLEANUP]
Do you wish to include the optional constituent SETUP?
4b. >No

5a. How many occurrences of <MAIN + [INTERFACE]> do you want?
5b. >3

6a. Do you wish to include the optional constituent CLEANUP?
6b. >LATER

LATER is a command understood by SPADE. It means that the user wants to terminate this path, but that it must be recommenced later. SPADE keeps track of conjunctive goals, reminding the user of the pending goal list if asked. Here it records the information required to return to, and eventually complete, the application of the SEQ rule. It can then go on to other goals.

Figure 7b. SPADE-0 Scenario (part b)

```
7a. Your pending goals are POLE, ... and WELL.  What NOW?
7b. >POLE
```

SPADE applies the SOLVE rule recursively.

```
8a. Rule for POLE is: SOLVE -> PLAN + [DEBUG]
      Rule for POLE-1 is: IDENTIFY | DECOMPOSE | REFORMULATE.
      What now?
8b. >IDENTIFY

9a. Rule for POLE-2 is: IDENTIFY -> PRIMITIVE | DEFINED.
      What now?
9b. >PRIMITIVE

10a. Rule for POLE-3 is: PRIMITIVE -> VECTOR | ROTATION | PENSTATE
      What now?
10b. >VECTOR
                .
                .
                .
     >FORWARD 100

11a. POLE completed.  Pending goals are: ROOF, WELL, and WW-17 (CLEANUP of
      WW).   What now?
11b. >WELL
                .
                .
                .

12a. Rule for WELL-1 is: PLAN -> IDENTIFY | DECOMPOSE | REFORMULATE.
      What now?
12b. >DECOMPOSE
```

Here we have substituted a grammar which contains rules for conjunction but not repetition. This allows us to illustrate the manner in which SPADE avoids interrogating the user when no actual decision is required.

```
13a. Rule for WELL-4 is: DECOMPOSE -> CONJUNCTION.
      (Forced.)
      Rule for WELL-5 is: CONJUNCTION -> LINEAR | NONLINEAR
      What now?
                .
                .
                .
```

skeleton for the superprocedure. (The goal of deciding whether to *include* the CLEANUP should not be confused with the goal of *designing* the CLEANUP once the need for it has been established.) Some users might prefer to defer this decision until the main steps have been further elaborated. SPADE should be able to accomodate the alternative solution order.

The typeout commencing at line 13a illustrates another feature of SPADE-0. (A similar sequence is shown at 2a.)

```
13a. Rule for WELL-4 is: DECOMPOSE -> CONJUNCTION.
     (Forced.)
     Rule for WELL-5 is: CONJUNCTION -> LINEAR | NONLINEAR
     What now?
```

Since the grammar is interpreted (rather than being "programmed in"), it is easy to try out alternative grammars. Suppose, as is shown here, we employ a simplified grammar in which the REPETITION rules have been eliminated. (This might be useful in tutoring a novice for example.) Then no decision is actually required in applying the DECOMPOSITION rule. SPADE should notice this, and not interrogate the user in such cases.

Figure 8 illustrates one possible derivation tree for WISHINGWELL as defined using SPADE-0. The utility of this record of the user's design decisions will become clearer when additional features of SPADE-0 are presented in the section on RAID.

The implementation of SPADE-0 (which is now in progress) will not be difficult. It is simply a bookkeeeping system for applying the planning grammar in generative mode to build a solution. The basic implementation technique is to provide an interpretive procedure for each grammatical operator (such as "|"). Additional features can be implemented by assigning specialist procedures to non-terminals of the grammar, as will be done for the debugging assistance illustrated later.

3.2. Towards SPADE-1, and Beyond

There is an upper bound on the utility of SPADE-0 which cannot be overcome by more careful human engineering. This is due to the fact that SPADE-0 does not have access to a description of the problem being solved. When application of the grammar rules results in a recursive application of SOLVE, SPADE-0 has no notion of the relationship of the subproblem to the top level goal. To overcome this fundamental limitation, we intend to design and implement SPADE-1.

Figure 8

ABBREVIATED HIERARCHICAL PLAN DERIVATION FOR A WISHINGWELL

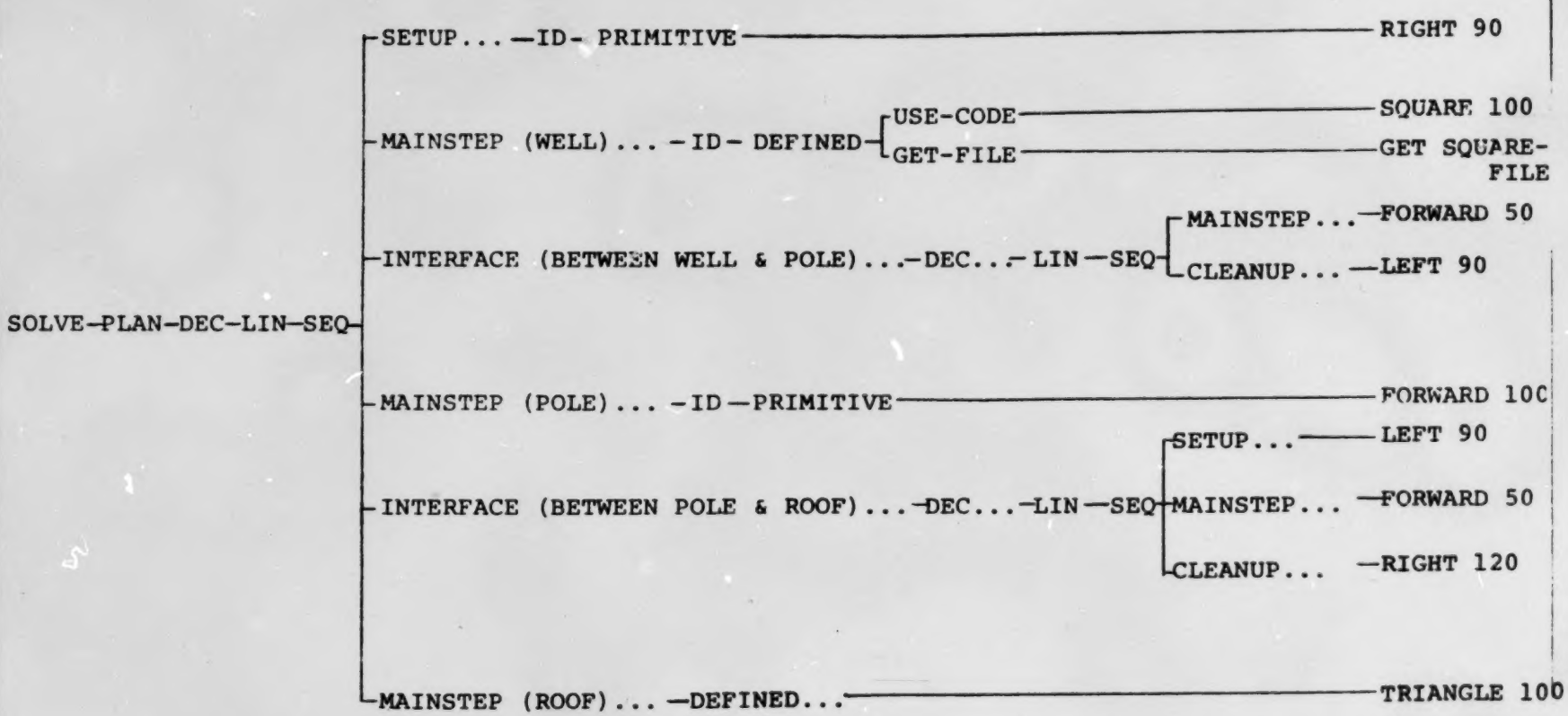


Figure 9 shows a hypothetical interaction with SPADE-1. In many respects, SPADE-1 will be similar to SPADE-0. It still is governed by a set of context free grammar rules, and still provides bookkeeping facilities for suspending and resuming subgoals. However, SPADE-1 requests that the user supply a formal description of the problem. (A library of standard problem descriptions is supplied for use as building blocks). The user need not comply with the request: however, without the problem description, SPADE-1 can help only as much as SPADE-0.

With a problem description, SPADE-1 would be able to provide additional assistance. It could notice when a procedure for solving a subproblem already exists in the answer library, by accessing the description of what that procedure accomplishes. It could perform rudimentary decompositions, and perform more substantial inferences when the user bypasses intermediate steps. Coupled with a performance annotation module (such as in [Goldstein 1974]), SPADE-1 could determine (in many cases) whether a given subprocedure satisfies its specifications.

The introduction of formal problem descriptions provides a first improvement over SPADE-0; introducing pragmatic constraints at choice points in the grammar would provide a second. This leads to a more elaborate linguistic formalism which we briefly consider in the concluding section.

While we plan to perform these extensions, from SPADE-0 to SPADE-1 and beyond, SPADE-0 will still serve a useful role. The grammar based editor can support experiments on several important issues: (1) the adequacy of the planning grammar; (2) the domain independence of the planning grammar; (3) the utility of the planning grammar in terms of structured programming considerations; (4) the pedagogical value of an articulate planning environment.

There is another dimension along which the theory as presented so far is incomplete. While the planning rules alone are sufficient to describe programs as static, finished objects, they do not capture the process of defining possible solutions, testing these solutions, and then debugging them. We need a theory of bugs, and of their diagnosis and repair, to supplement our theory of plans. This is the purpose of the next two sections.

Figure 9. SPADE-1 Scenario

With a description of the problem, SPADE could be more helpful.

1a. Tell me the name of your toplevel procedure?
1b. >WW

2a. Would you like to define a model?
2b. >WISHINGWELL

3a. OK. WISHINGWELL model found in Model Library.
Rule for WW-1 is: PLAN -> IDENTIFY | DECOMPOSE | REFORMULATE.
What now?
3b. >ROOF POLE WELL

4a. These subgoals correspond to model parts. Are you pursuing a LINEAR DECOMPOSITION plan?
4b. >YES

5a. The pending subgoals are: ROOF, POLE, WELL and the decisions regarding inclusion of preparatory steps (WW-5) for WW. What now?
5b. >ROOF

6a. In the WW model, ROOF is described as a triangle. There is a TRIANGLE procedure in the answer library. Hence, an IDENTIFICATION plan may succeed. What now?
6b. >IDENTIFY

7a. The TRIANGLE procedure in the answer library satisfies the intrinsic description of ROOF. Would you like to use it?
7b. >YES

8a. OK. The ROOF is complete. Your pending goals are POLE, WELL, and WW-5. What now?
8b. >

.
.
.

9a. The first order goals for the parts are now complete. WW-5 (the choice of preparatory steps for WW) is complete. You have not expanded the definitions for the interface step, WW-6, nor for the cleanup step, WW-8. What now?

.
.
.

4. A Grammatical Theory of Debugging

> Bugs are so important that it is useful to classify them and give the classes names. In real world problem solving we often give names to important classes of bugs. In electrical engineering, for example, one class of bug is "instability." It may be manifest as "thermal runaway" or "spurious oscillation" in an amplifier. The underlying cause is "positive feedback," and there are several possible cures (patches) which may be applied: "negative feedback," or "isolation," for example.
>
> [Sussman, 1973, p. 170.]

In earlier sections, we constructed a grammar of planning concepts and described programs as the terminal strings generated by this grammar. Unfortunately, problem solvers, whether human or machine, must often decide on a plan despite not only knowledge which is incomplete or uncertain, but also limitations on time and memory resources. The best of choices in such situations can turn out wrongly: debugging is then required. In this section, we follow Sussman's advice, developing a classification of bugs. Our goal in this classification scheme is to unify our approaches to planning and debugging by tracing the origin of bugs to various types of erroneous planning choices. In section five, we apply this perspective on possible planning errors to the design of a debugging assistant called RAID to be incorporated into the SPADE environment.

4.1. Types of Bugs

Given our perspective on planning, debugging can be analyzed as the localization and repair of errors in applying the grammar rules during generation. Since our planning rules were constructed from operators for conjunction, for disjunction and for optionality, there arise three basic classes of error:

(1) *syntactic bugs* in which the planning grammar is violated, such as when a required conjunct is missing.

(2) *semantic bugs* in which the plan is syntactically well-formed but some semantic constraint arising from the particular problem is violated, such as when a syntactically optional constituent, needed because of the semantics of the particular problem, is missing.

(3) *pragmatic bugs* in which an inappropriate selection from a set of mutually exclusive disjuncts is made.

This categorization is not complete: two other classes of bugs are "circumlocutions" and "slips of the tongue."[13] The first class represents plans which are successful but inefficient. The second class refers to miscellaneous errors in execution including mis-typings, mis-spellings and incorrect programming language syntax that do not reflect basic conceptual mistakes in the plan.

4.2. Syntactic Planning Bugs

When a decision made during a problem solving session violates the planning grammar, the resultant bug is termed *syntactic*.[14] An example of a syntactic bug is failure to include an obligatory conjunct. To illustrate this, consider the following error. In the solution of a problem, one subgoal matches a previously solved problem. Hence, the problem solver incorporates a call to the appropriate subroutine into the solution. But it is common to forget to load the file containing the subroutine into the current workspace. Figure 10 illustrates this difficulty: as before, the goal is to write a program that draws a wishingwell. The roof is a triangle, which corresponds to a previously defined subprocedure. A call to TRIANGLE is placed in the WW procedure, but WW is executed before the file containing TRIANGLE is loaded.[15]

In terms of our planning grammar, this is a *syntactic bug*. The WW procedure is ungrammatical. The appropriate rule describing this situation is:

P4: DEFINED -> USE-CODE & GET-FILE

but the file retrieval is missing.

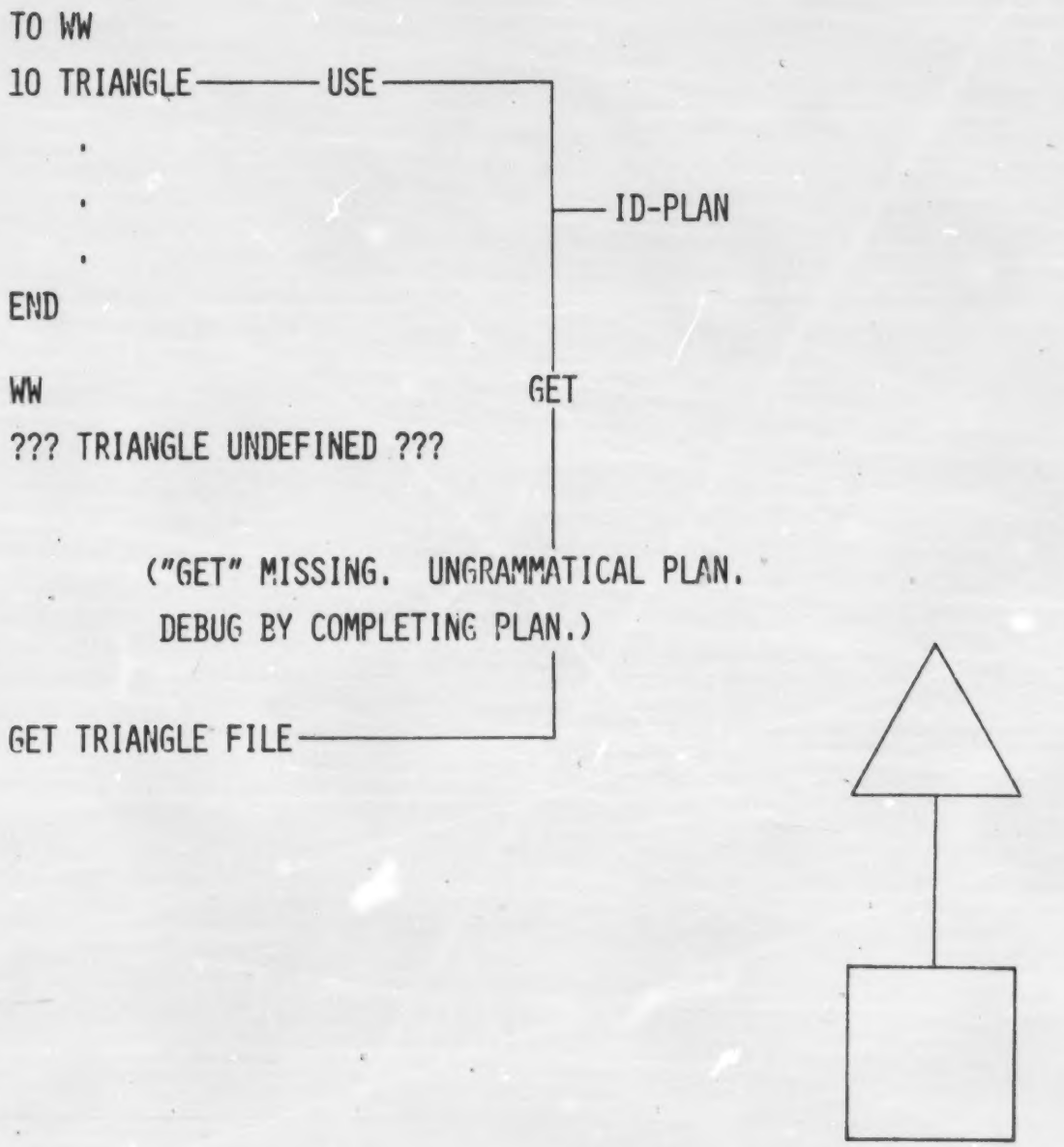
Thus, syntactic bugs are those in which a necessary conjunct of a planning rule is not present in the plan. (Syntactic bugs might also be caused by the presence of an illegal extra constituent, but this class of problems seems less common.) Normally one would not expect a machine problem solver to make this kind of error, given a correct planning theory and no heuristic limitations. However, resource limits on time or space might result in this performance. Moreover, it is a common human error.[16]

The basic technique for *repairing* a syntactic bug (once isolated) is to redo the culpable planning decision in such a way that the grammar is no longer violated. For the case of a missing but obligatory conjunct, this implies solving for the constituent in question, and incorporating that solution into the larger solution at the required point. For the WW example in particular, it means getting the TRIANGLE procedure from a file, and then re-executing WW in the corrected environment.

Figure 10

DEBUGGING A SYNTACTICALLY INCORRECT PLAN

A NECESSARY CONJUNCT IS MISSING



the intended picture

4.3. Semantic Planning Bugs

Semantic bugs differ from syntactic bugs in that no planning decision violates the underlying grammar; rather the usual case is that a constituent which is optional in the grammar is not present, but is needed due to the semantics of the particular problem. This distinction can be understood more clearly by considering that *syntax* supplies broad constraints on the structure of solutions to all problems; *semantics* supplies additional constraints in terms of features of the particular problem at hand. Rules P1 and P8 are typical rules in the grammar for which this kind of difficulty can arise:

P1: SOLVE -> PLAN + [DEBUG]
P8: SEQ -> [SETUP] + <MAINSTEP + [INTERFACE]>* + [CLEANUP].

Debugging is necessary if the program produced during planning fails to accomplish its intended goals; otherwise, debugging is unnecessary. For a concrete example involving P8, let us return to the WW problem. Part of the problem specification is that the wishingwell be drawn in an upright position. Suppose that the order in which the main steps are executed is to be: ROOF, POLE, and then WELL. The subprocedure for the TRIANGLE expects the turtle to begin at a vertex, oriented along the circumference. Therefore, an initial SETUP (syntactically optional) rotation is required to vertically orient the wishingwell as a whole. Furthermore, additional interface steps are required to establish the required relationship between the ROOF and the POLE, that the POLE connect to the ROOF by intersecting with the center of its bottom side. Figure 11 illustrates this local geometry, contrasting a semantically incomplete WW program to a corrected version.

Since it is often an effective heuristic to design main steps before interfaces, one would not be surprised if a human programmer designed the subprocedures for the roof, pole and well, and then concatenated them, but forgot to include these necessary interfaces. Moreover, even for a machine problem solver, there are situations in which it would be more efficient (and therefore rational) to determine the need, if any, for such interface steps via trial execution and debugging, than via thorough but resource-intensive initial planning.

In terms of the planning grammar, the overall plan for the WW is described as a sequential plan -- that is, a sequence of main steps for the parts with optional interfaces. Given rule P8, the WW program illustrated by the previous figure is syntactically acceptable, but semantically incomplete.

Semantic bugs can also occur when an optional constituent is present, but

Figure 11

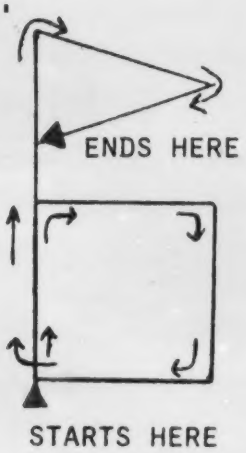
DEBUGGING A SEMANTICALLY INCORRECT PLAN

AN OPTIONAL CONJUNCT IS MISSING

FOR EXAMPLE:
"WW" MISSING INITIAL SETUP,
AND INTERFACE FOR POLE.

```
TO WW
10 WELL ——— MAINSTEP ┐
20 POLE ——— MAINSTEP ┘— SEQ-PLAN
    :
    :
    :
```

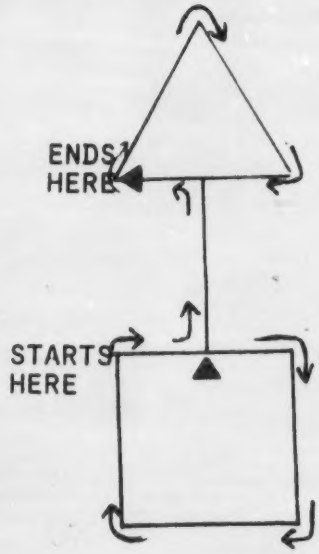
★ USE IMPERATIVE KNOWLEDGE OF MODEL PREDICATES TO COMPUTE MISSING STEPS.



THE SEMANTICALLY CORRECTED PROCEDURE

```
TO WW
★5  WW-SETUP ———— SETUP ┐
10  WELL ———————— MAINSTEP ┐
★15 WELL-POLE-INTER — INTERFACE ┐
20  POLE ——————— MAINSTEP ┘— SEQ PLAN
    :
    :
    :

TO WW-SETUP
10 RIGHT 90 ┐
20 FORWARD 50 ┘— SEQ — BODY OF SETUP STEP
30 RIGHT 90
END
    :
    :
    :
```



semantically inappropriate. An example which we observed in a high school student was to always begin a procedure with the PENUP command, even when the first main step was to draw a visible vector. This resulted in either: (a) when the program was first run, the first vector would be missing, and then the PENUP would be deleted by a debugging edit; or (b) a PENDOWN command would be added to the procedure: inefficient but otherwise harmless extra steps.

The general repair strategy for semantic bugs is to redo the culpable planning decision in such a way as to satisfy the violated semantic constraints. In particular the repair for a semantically incomplete plan is to solve for the missing conjuncts and incorporate them into the solution as a whole. For the wishingwell, this involves designing setup and interface steps, and then editing the WW superprocedure to employ them.

4.4. Pragmatic Planning Bugs

Some grammar rules describe alternative strategies to accomplish a given plan. Formally these appear as mutually exclusive disjuncts. Examples include:

```
P2:  PLAN          -> IDENTIFY | DECOMPOSE | REFORMULATE
P3:  IDENTIFY      -> PRIMITIVE | DEFINED
P6:  CONJUNCTION   -> LINEAR | NONLINEAR
P15: REPETITION    -> ROUND | RECURSION
```

Pragmatic bugs are those in which an incorrect disjunct is chosen.

As an illustration, consider grammar rule P6 for conjunctive plans. It specifies two alternatives for accomplishing a set of subgoals: a linear and a nonlinear strategy. Now in this case, the formal roles played by the alternative disjuncts are syntactically indistinguishable with respect to the overall grammar. The *pragmatic* difference, which is not formalized here, is that a linear decomposition solves for the sub-problems independently while a nonlinear decomposition solves for some subgoals given knowledge of other subgoals.

In general, linear plans are simpler to apply because of their independence assumption. However, pragmatic bugs arise when the planner is faced with a type of problem in which there are inherent interactions between the steps. An example of where linear problem solving is inadequate in the graphics world is the apparently simple task of drawing a square *inside* a triangle (figure 12). Suppose a linear plan is pursued. This gives rise to two main steps (the square and the triangle), and an interface step. If the main steps are solved independently of one another (by means of SQUARE and TRIANGLE subprocedures), it is likely that the figures produced will be of the wrong size to permit the desired INSIDE relation to hold. This violation cannot be corrected by altering

Figure 12

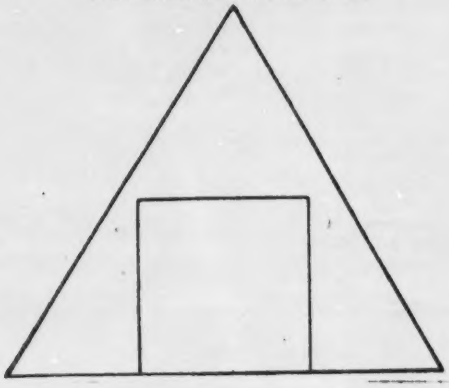
DEBUGGING A PRAGMATICALLY INCORRECT PLAN

AN INCORRECT DISJUNCT HAS BEEN SELECTED

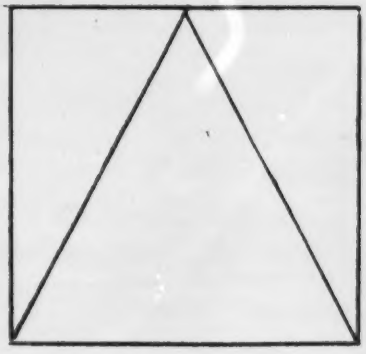
```
TO SQUARE-INSIDE-TRIANGLE
10 SQUARE
20 TRIANGLE
END
```

LINEAR PLAN --
SQUARE AND TRIANGLE
DESIGNED
INDEPENDENTLY.

INTENDED PICTURE:



ACTUAL PICTURE:



DEBUG BY CHANGING TO NON-LINEAR PLAN.
DESIGN SQUARE IN THE CONTEXT OF TRIANGLE.

the order of composition; nor can it be repaired by modifying the interface. The bug is pragmatic, in that neither syntax nor semantics are violated, but the choice of the linear over the nonlinear disjunct nevertheless leads to an unsuccessful plan.

A pragmatic bug is repaired by redoing the culpable planning decision so as to satisfy the violated pragmatic constraint. (It may be that the problem solver was ignorant of the relevant constraint prior to solving the current problem. This brings up the matter of skill acquisition which is deferred till the concluding section.) In the SQUARE-WITHIN-TRIANGLE problem, violation of the predicate INSIDE is repaired by changing to a non-linear plan. The second main step to be solved must be designed in the context of a particular size decision for the first main step. For example, the specification for TRIANGLE is changed to require that its side be larger than a constant which is determined by the side length of SQUARE.

4.5. "Circumlocutions" (Inefficiency Bugs)

A procedure which solves its specified problem, but in a roundabout manner, is said to have a "circumlocution" or an *inefficiency bug*. Such inefficiencies can occur in plans where a non-optimal disjunct is chosen or an unnecessary (but harmless) optional constituent is included. Correcting inefficiencies is the typical concern of compiler theory and we do not address it here, except to make the point that the hierarchical annotation (or derivation) generated by the grammar is conceivably a useful description for a compiler to access.

To illustrate this, consider *rational form violations*, the subclass of inefficiencies due to local oddities in the code, such as sequential invocations of a given primitive.[17] This class of inefficiencies has been extensively investigated in the literature on optimizing compilers. However, it is possible that such a rational form violation is due to some serious omission in the program; i.e., it is a warning that a bug may exist [Goldstein 1974]. Traditional compilers have no basis for a judgment, but access to the planning derivation of the program can often illuminate this issue.

For example, one of the ways in which such an inefficiency bug can arise is from the use of an "evolutionary" plan [Miller 1976]. Although the grammar provided in this paper does not attempt to formalize this type of plan, basic evolutionary plans are not complex. The programmer attempts to alter the code of a previous program to achieve the specifications of a new, but similar, problem. To illustrate such a situation, however, we must develop a somewhat elaborate example. Please reexamine figure 5. A wishingwell, initially viewed as involving three subproblems, has been reformulated so as to involve two main

steps, the TREE and the WELL. The TREE program is *state transparent*: it leaves the turtle in the same state in which it started, at the bottom of its TRUNK (which serves as the POLE of WW). WW incorporates a nonlinearity for efficiency: the top side of the WELL is accomplished in two parts, to avoid retracing previous vectors.[18] Suppose that the programmer needs a SQUARE subprocedure for use in another project. One strategy is to adapt WW by deleting the call to TREE (figure 13). After this deletion, though, the resulting SQUARE contains sequential calls to FORWARD: a rational form violation. The optimization is to combine these two invocations into a single call to the FORWARD primitive.

Thus, a compiler could first check whether an evolutionary plan governs the inefficiency. If so, it could perform the optimization with some confidence. If not, it should notify the programmer of the oddity in the code.

4.6. "Slips of the Tongue" (Execution Errors)

A final category of bugs is necessary when human programming protocols are to be analyzed. This class, "slips of the tongue," is a catch-all for typographical errors, confusions due to orthographic similarity, incorrect programming language syntax, noise on the computer line, and other failures to successfully type in a statement of code. They are often diagnosed by conventional computing environments, simply as a result of the code being unrecognizable. The plan is not affected. We include this class for completeness, so that our discussions may span the space of possible bugs. The planning grammar does not provide an explanation for the origins of these bugs.[19]

The general repair technique for slips of the tongue is to: (a) undo the side effects, if any, of the incorrect type-in; and (b) reexecute the type-in correctly in the restored environment. This could be captured by a rule such as:

REPAIR -> [UNDO] + REDO

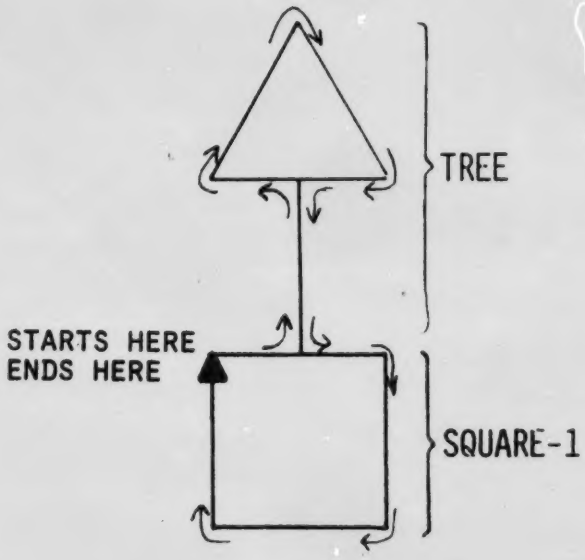
A common error in debugging technique is to compound an initial "slip of the tongue error" by reexecuting, without undoing undesirable side effects.[20]

Having a classification of basic bug types does not solve the debugging problem: it is only a starting point. The next step is to develop a theory of diagnosis and repair, by which the underlying bug made manifest by an unsuccessful program run can be diagnosed, and then repair knowledge associated with this bug type can be applied to correct the program. Section five designs the RAID assistant that will monitor a programmer during the planning of a procedure and generate caveats regarding possible errors for aid in subsequent debugging. This monitoring will happen within the SPADE editing environment.

Figure 13

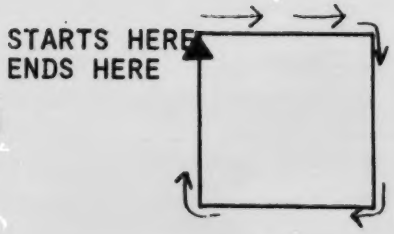
DEBUGGING A CIRCUMLOCUTION OR INEFFICIENT PLAN

```
TO WW
5  RIGHT 90
10 FORWARD 50
20 TREE
30 FORWARD 50
40 RIGHT 90
50 FORWARD 100
60 RIGHT 90
70 FORWARD 100
80 RIGHT 90
90 FORWARD 100
END
```



⬇ EVOLUTIONARY PLAN

```
TO SQUARE-1
5  RIGHT 90
10 FORWARD 50 ┐
30 FORWARD 50 ┘ RATIONAL FORM VIOLATION
40 RIGHT 90
     :
     :
END
```



⬇ CAVEAT DRIVEN DEBUGGING

```
TO SQUARE-2
5  RIGHT 90
10 FORWARD 100
40 RIGHT 90
     :
     :
END
```

5. The RAID Debugging Assistant

> Let us focus on one particular component of [general heuristic knowledge]: the art and techniques of ... *debugging*. The school experience is dominated by the normative attitude implied by "right answer vs. wrong answer". The mathematician's experience of mathematics is dominated by the purposeful-constructive attitude implied by the struggle to "make it work". He abandons an idea not because it happened to go wrong, but because he has understood that it is unfixable. Dwelling on what went wrong becomes a source of power rather than a piece of masochism (as it would appear to most fifth graders in traditional math classes).
>
> [Papert, 1973, p. 10]

5.1. Diagnosis and Repair

We have now developed a taxonomy of bug types -- of what use is it? Its first use, we believe, is that it clarifies our understanding of debugging by identifying major categories of error. Secondly, it suggests how to design better debugging aids for the programmer and problem solver. In this section, we develop this position by designing the RAID component of the SPADE program editor. *RAID* is an acronym for *RAtional Implementation of Debugging*, stressing our belief that debugging is often a consequence of *heuristically justifiable* problem solving, not an embarrassment indicative of irrational or "sloppy" thinking. RAID is a tool designed to make debugging a source of power to the problem solver, as Papert suggests it can be.

Let us consider further how the taxonomy clarifies our understanding of debugging. A programmer's approach to debugging is, naturally, colored by the diagnostic tools provided by the particular computer system. However, the facilities provided by a wide range of computing environments have much in common. These tools manifest what we term *surface debugging techniques*. They are based on examination of the code and snapshots of the computational process elicited by the code, both relatively superficial descriptions of the procedure as conceived by the programmer. Figure 14 shows a grammar which partially formalizes this surface debugging activity.

Access to the problem description and -- most importantly -- the programmer's *plan* allows for a deeper analysis of debugging strategies. Figure 15 shows a taxonomy of these debugging strategies. Figure 16 shows how this taxonomy is transformed into what we term a *deep debugging grammar*, for contrast with the previous grammar. Notice that examination of the plan plays an important role.

Figure 14. A Surface Grammar For Debugging

```
DEBUG      -> <[DIAGNOSE] + [REPAIR]>*
DIAGNOSE   -> <ASK | TRACE | "error">*
TRACE      -> [SELF-DOC*] + RUN*
SELF-DOC   -> ADD-PAUSE | ADD-PRINT | ADD-TRACE
ASK        -> "print definition" | "print value" |"print file"| ...
REPAIR     -> <RUN | EDIT | SOLVE>*
ADD-PAUSE  -> ADD
ADD-PRINT  -> ADD
ADD-TRACE  -> ADD
EDIT       -> ADD | DELETE | CHANGE
RUN        -> "run statement of code" + "response" + [DEBUG]
ADD        -> "add statement of code" + "response" + [DEBUG]
DELETE     -> "delete statement of code" + "response" + [DEBUG]
CHANGE     -> "change statement of code" + "response" + [DEBUG]
```

FIGURE 15 - A TAXONOMY OF DEBUGGING TECHNIQUES

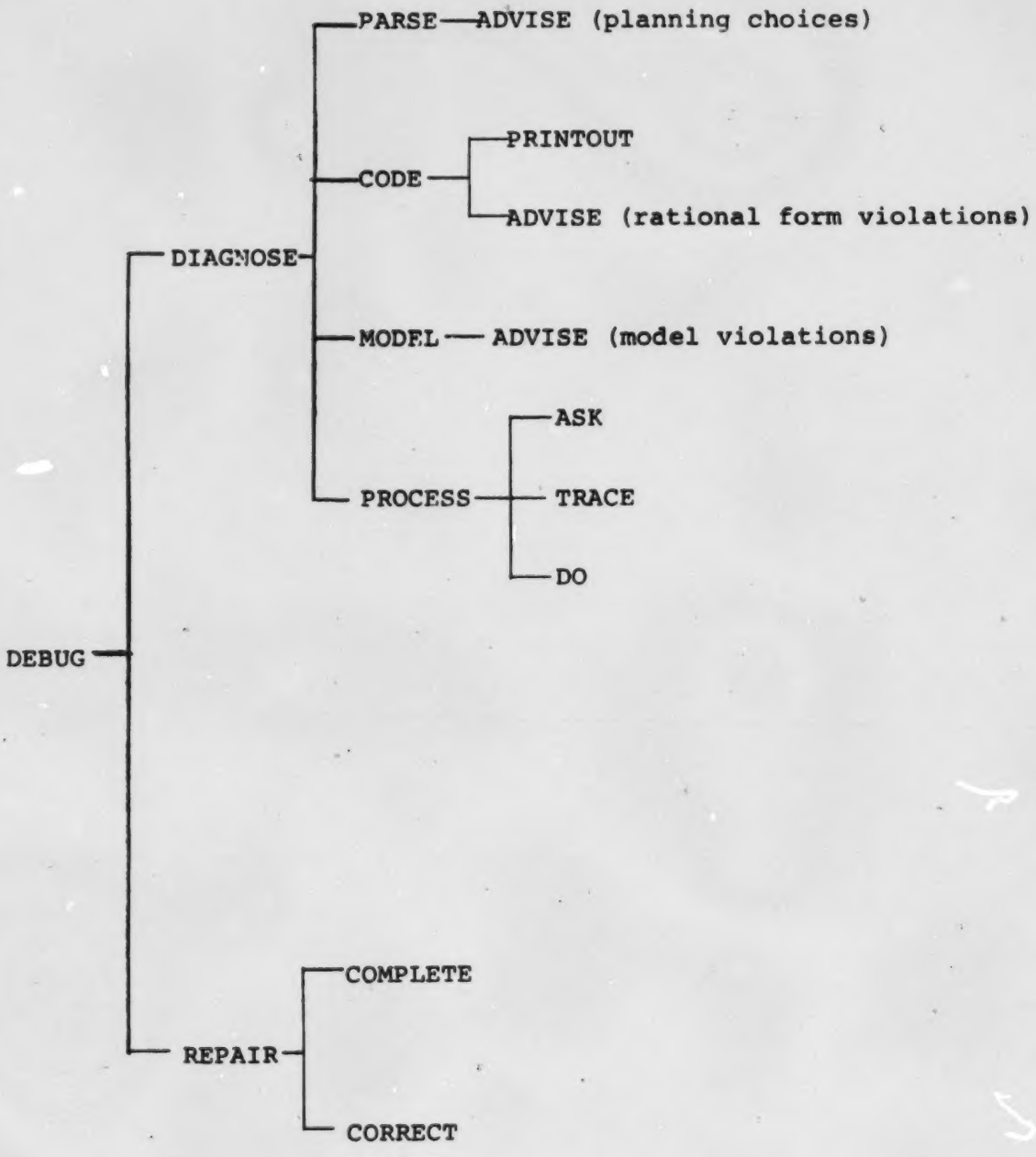


Figure 16. A Deep Grammar For Debugging

```
DEBUG     -> <[DIAGNOSE] + [REPAIR]>*

DIAGNOSE -> <PARSE | CODE | MODEL | PROCESS>*

PROCESS  -> ASK | TRACE | DO

CODE      -> PRINTOUT | "advise rational form violations"

MODEL     -> "advise model violations"

PARSE     -> "advise heuristic planning choices"

REPAIR    -> COMPLETE | CORRECT

COMPLETE -> "solve for missing conjunct"

CORRECT  -> "choose alternative disjunct"
```

In the SPADE system, the end product of the interaction is not merely a program, but a program annotated by its associated plan derivation (please refer to figure 8 presented earlier). The reader has undoubtedly noted that far more interaction would be necessary with SPADE, than with an ordinary editor.[21] In return for this extra planning effort, there are several potential benefits. The first is that by knowing the plan, the RAID component of SPADE would generate caveats regarding possible bugs for aid in subsequent debugging. Since definition of the program generally occupies far less time than debugging, some additional effort in planning may well be worthwhile in terms of more efficient debugging. It is also possible that articulating the plan serves to improve a student's planning skills.[22] Finally, the plan as commentary should make the resulting code far more understandable to other programmers who, in large projects, may be expected to modify or debug the package. We have yet to consider "human-engineering" aspects in designing SPADE/RAID, nor have we begun to experiment with it. Here, our goal is only to describe those parts of the RAID debugging assistant that are predicated on our taxonomy of bug types.

5.2. Aid In Diagnosing Syntactic Bugs

SPADE provides the facility of being able to suspend the construction of a solution of one sub-problem in order to analyze other goals. This is useful, since occasionally insight into the solution of other goals is helpful for completing the current problem. SPADE provides bookkeeping facilities for this suspension. The goal can be resumed later. To illustrate this, we shall refer to figure 17. Line 3b shows the user choosing to postpone a goal.

```
3a. Rule for POLE-2 is: IDENTIFY -> PRIMITIVE | DEFINED.
      What now?
3b. >LATER.
```

RAID, the debugging component, would create a caveat each time this happens. The caveat would record the fact that the plan is syntactically incomplete at this point. A *caveat* is a demon, which would triggered by premature execution of the procedure being designed. If this occurs, the function of the caveat would be to remind the user of the syntactic incorrectness of the program.

```
5b. >RUN WISHINGWELL

6a. Running WISHINGWELL at this time results in the bug manifestation:
                UNDEFINED PROCEDURE POLE.
    This is because WISHINGWELL is syntactically incomplete: it contains
    the pending subgoal POLE-2. The recommended repair technique is to
```

Figure 17. RAID Advice for Syntactically Incomplete Planning Bugs

1a. Which subgoal would you like to pursue?

1b. >POLE

2a. Rule for POLE-1 is: PLAN -> IDENTIFY | DECOMPOSE | REFORMULATE.
What now?
2b. >IDENTIFY

3a. Rule for POLE-2 is: IDENTIFY -> PRIMITIVE | DEFINED.
What now?
3b. >LATER.

The LATER command causes further processing of this subgoal to be postponed. RAID, the debugging component, creates a caveat each time a LATER command occurs. The caveat records the fact that this part of the plan is syntactically incomplete at this point.

4a. POLE-2 postponed. Your pending goal list contains the remaining subgoal WELL. Which goal would like to pursue?
4b. >WELL

.
.
.

5a. We have now completed the WELL subgoal. What now?
5b. >RUN WISHINGWELL

The caveat recorded earlier is a "demon," triggered by premature execution of the procedure being designed. If this occurs, the function of the caveat is to remind the user of the syntactic incorrectness of the program.

6a. Running WISHINGWELL at this time results in the bug manifestation:
UNDEFINED PROCEDURE: POLE.
This is because WISHINGWELL is syntactically incomplete: it contains the pending subgoal POLE-2. The recommended repair technique is to complete the solution of the subgoal POLE-2. Would you like to work on POLE-2?
6b. >Yes.

7a. You were solving POLE via an IDENTIFICATION plan. You postponed the decision (POLE-2) as to which answer library to use.
The rule for POLE-2 is: IDENTIFY -> PRIMITIVE | DEFINED.
What now?

.
.
.

complete the solution of the subgoal POLE-2. Would you like to work on POLE-2?

Line 6a in the figure illustrates this.

5.3. Aid in Diagnosing Semantic Bugs

Whenever an optional constituent is rejected, RAID would create a caveat to the effect that the plan may be semantically incorrect at this point. When the program subsequently is executed and bugs occur, the programmer could request aid. This aid would include a description of which planning decisions are possible semantic errors. This sort of interaction with SPADE is illustrated by figure 18.

Line 1b shows the programmer choosing to dispense with an optional interface step.

1a. Is there an INTERFACE following WELL?
1b. >NO

When the program is subsequently tested at line 2b, the programmer is not satisfied with the results.

4b. >ADVISE-PLAN

5a. WELL may be semantically incomplete. Perhaps an INTERFACE is needed after WELL. Would you like to reconsider, and include such a step?

RAID is then depicted pointing out that the decision to not include an optional interface may be the culprit (line 5a).

5.4. Aid in Diagnosing Pragmatic Bugs

Certain alternative planning methods have the relationship that one method is simpler but subject to failure in some cases, while the other is more complex but more powerful. Examples are LINEAR versus NONLINEAR plans and ROUND versus RECURSIVE plans. Whenever the user applies the simpler of one of these pairs, RAID creates a caveat to the effect that a commitment to a possibly incorrect plan has been made. This caveat serves the same purpose as those created above for possible semantic bugs. Upon subsequent execution, if the user is unhappy with the results, aid can be requested. The caveat then points out possible pragmatic errors.

Figure 18. RAID Advice For Semantically Incomplete Planning Bugs

1a. Is there an INTERFACE following WELL?
1b. >NO

Whenever an optional constituent is rejected, the RAID debugging component creates a caveat to the effect that the plan may be semantically incomplete at this point.

2a. We have now completed subgoal WELL. What now?
2b. >RUN WISHINGWELL

When the program subsequently is executed and bugs occur, the programmer can request aid. This aid includes a description of which planning decisions are possible semantic errors. In this case, the WW procedure produces a picture which does not satisfy the WISHINGWELL model. While SPADE-0 can aid in plan diagnosis, it cannot detect model violations without user input. (SPADE-1, however, would perform model diagnosis.)

3a. OK. What now?
3b. >DIAGNOSE

4a. Rule is:
DIAGNOSE -> ADVISE-PLAN | ADVISE-CODE | ADVISE-MODEL | ADVISE-PROCESS.
What now?
4b. >ADVISE-PLAN

5a. WELL may be semantically incomplete. Perhaps an INTERFACE is needed after WELL. Would you like to reconsider, and include such a step?
5b. >YES

6a. Solving for WELL-13 (INTERFACE after WELL).
Rule is: SOLVE -> ...

Figure 19 illustrates this kind of interaction with SPADE.

5b. >ADVISE-PLAN

6a. In designing SQUARE-WITHIN-TRIANGLE-3, you opted for a LINEAR decomposition. It is possible that this problem involves some interaction between TRIANGLE and SQUARE. Do you wish to reconsider your previous decision, and try a NONLINEAR decomposition?

Line 6a in the figure shows the RAID component alerting the user to a possible pragmatic planning bug.

5.5. Assistance in Repair

The system could do more than just alert the user to the problem. It could also (a) return the user to the suspended goal, and (b) inform the user, by means of the grammar, of what alternative constituents are available. Line 7a of figure 17 (presented earlier) illustrates this repair assistance for the case of a syntactic bug.

7a. You were solving POLE via an IDENTIFICATION PLAN. You postponed the decision (POLE-2) as to which answer library to use.
The rule for POLE-2 is: IDENTIFY -> PRIMITIVE | DEFINED.

Suppose the user decides to undo a given planning decision, perhaps abandoning a very detailed plan which resulted from considerable effort, in favor of a new approach. It is possible that later the user may reconsider, and wish to reactivate the abandoned approach. It would be rather unfortunate if RAID had erased all records of its existence. In fact, it would not. In order to keep track of both active and hung alternative versions of each planning decision, however, the derivation *tree* representation ceases to be adequate. A more elaborate data structure is required; SPADE/RAID would use a *chart* for this purpose. The chart data structure was introduced in computational linguistics [Kay 1973; Kaplan 1973] to economically store a record of alternative well-formed substrings discovered during sentence parsing. We introduce the chart here for similar reasons. An example of such a chart applied to the planning context is shown in figure 20; figure 21 shows a user taking advantage of this capability.

The chart data structure provides benefits which feed back into the planning assistant as well. For example, with the chart one can distinguish between using a *copy* of a previous subprocedure "in line," versus using the previous subprocedure as a "black box" which is invoked:

Figure 19. RAID Advice For Pragmatically Incorrect Planning Bugs

```
1a. Rule for SQUARE-WITHIN-TRIANGLE-1 is:
              PLAN -> IDENTIFY | DECOMPOSE | REFORMULATE
    What now?
1b. >CONJ

2a. I have assumed DECOMPOSE -> CONJUNCTION.
    Rule for SQUARE-WITHIN-TRIANGLE-3 is:
              CONJUNCTION -> LINEAR | NON-LINEAR.
    What now?
2b. >LINEAR
```

Certain alternative planning methods have the relationship that one method is simpler but subject to failure in some cases, while the other is more complex but more powerful. Examples are LINEAR versus NONLINEAR plans and ROUND versus RECURSIVE plans. Whenever the user applies the simpler of one of these pairs, RAID creates a caveat to the effect that a commitment to a possibly incorrect plan has been made.

```
3a. Rule is ...
3b. >

    .
    .
    .

4a. We have now completed SQUARE-WITHIN-TRIANGLE.  What now?
4b. >RUN SQUARE-WITHIN-TRIANGLE
```

Upon subsequent execution, if the user is unhappy with the results, aid can be requested. The caveat then informs him of possible pragmatic errors. Here, the SQUARE-WITHIN-TRIANGLE procedure draws the picture "inside out."

```
5a. Rule is:
    DIAGNOSE -> ADVISE-PLAN | ADVISE-CODE | ADVISE-MODEL | ADVISE-PROCESS.
    What now?
5b. >ADVISE-PLAN

6a. In designing SQUARE-WITHIN-TRIANGLE-3, you opted for a LINEAR
    decomposition.  It is possible that this problem involves some
    interaction between TRIANGLE and SQUARE.  Do you wish to reconsider
    your previous decision, and try a NONLINEAR decomposition?
6b. >YES

7a. What non-linear constraint(s) do you wish to add to the SQUARE?
7b. >

    .
    .
    .
```

FIGURE 20 - A CHART OF ALTERNATIVE PLANS

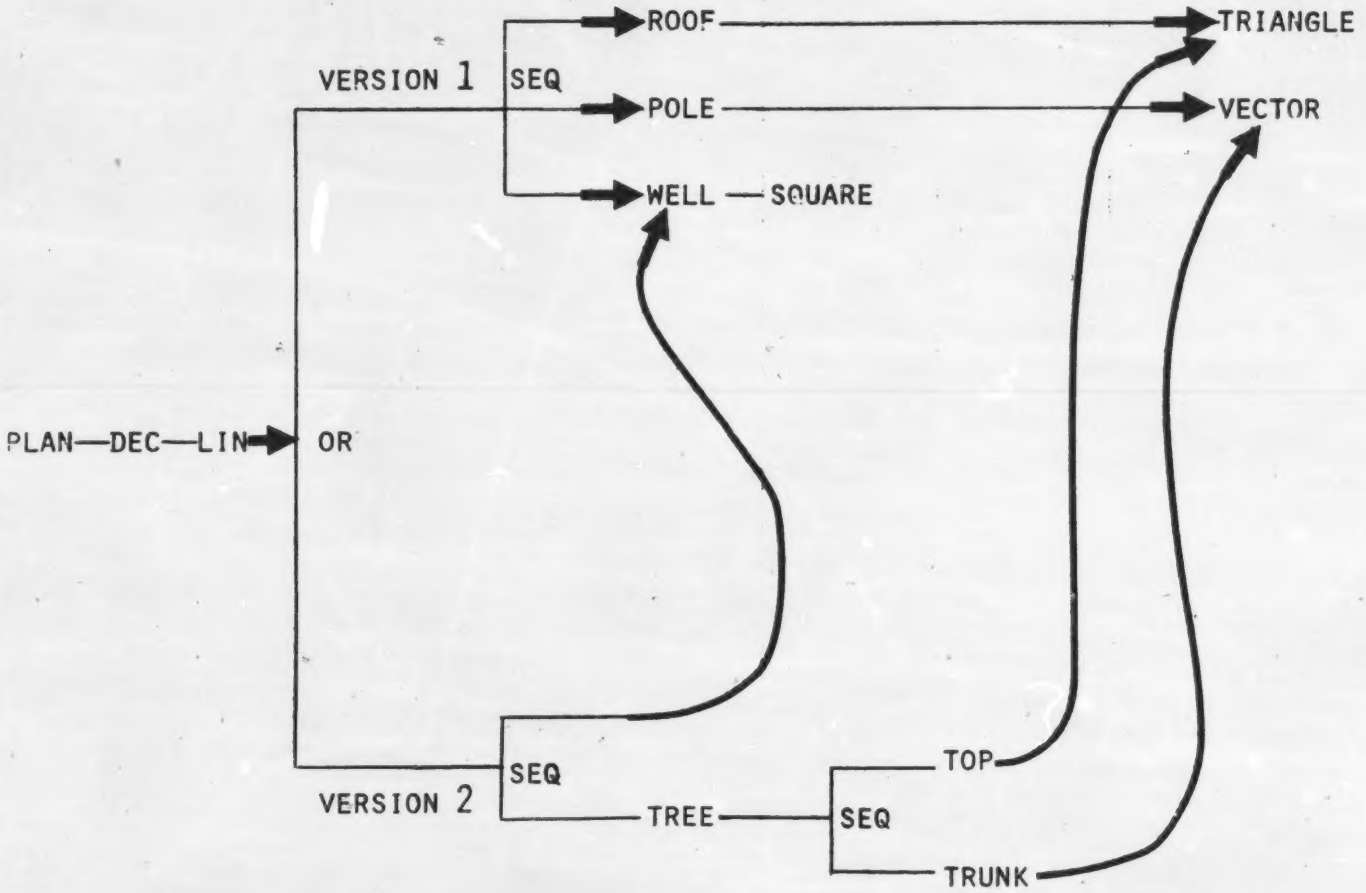


Figure 21. Returning to a Previously Hung Plan

```
1a. Rule for WW-1 is:  PLAN ->  IDENTIFY | DECOMPOSE | REFORMULATE
      What now?
1b. >DECOMPOSE
                .
                .
                .

2a. What are your subgoals?
2b. >ROOF POLE WELL
                .
                .

                .

3a. Rule for POLE-3 is: IDENTIFY -> PRIMITIVE | DEFINED
      What now?
3b. >REDO WW-1
```

Considerable planning effort has been invested in the current version of WW-1. The system should be able to record the state of the work so far in case it is ever needed again. It might seem too costly, though, to save away multiple copies of the plan derivation every time a choice is redone...

```
4a. Previous choice on WW-1 undone.
      Rule for WW-1 is: PLAN -> IDENTIFY | DECOMPOSE | REFORMULATE
      What now?
4b. >REFORMULATE
                .
                .
                .

5a. What are your subgoals?
5b. >TREE WELL
                .
                .
                .

6a. Rule is: SEQ -> [SETUP] + <MAINSTEP + [INTERFACE]>* + [CLEANUP]
      What now?
6b. >REDO WW-1
```

At this point, the user has decided to return to the previous approach, finding that the REFORMULATION has resulted in an even harder problem. SPADE's chart representation for plan derivations in fact allows prior versions to be saved, without incurring overwhelming storage costs.

```
7a. A previously hung version of WW-1 (WW-1:V1) exists, involving a
      DECOMPOSITION plan.  Do you wish to reactivate it?
7b. >YES

8a. When WW-1:V1 was hung, your current goal was POLE-3.
      Rule for POLE-3 is: ...
```

P4': DEFINED -> COPY-CODE|CALL-CODE & GET-FILE

The revised rule P4' has the virtue that one can specify whether *future* changes to the existing subprocedure should affect the procedure currently being solved. If the CALL-CODE disjunct is chosen, the chart will contain only a *pointer* to the shared substructure: future improvements in the subprocedure will also benefit the current procedure. Conversely, future changes could introduce unanticipated perturbations. This indicates how the insights gained from a grammatical approach to problem solving can lead to formalizing the origins of yet another commonly observed source of program bugs.

6. Conclusions

6.1. Limitations and Extensions

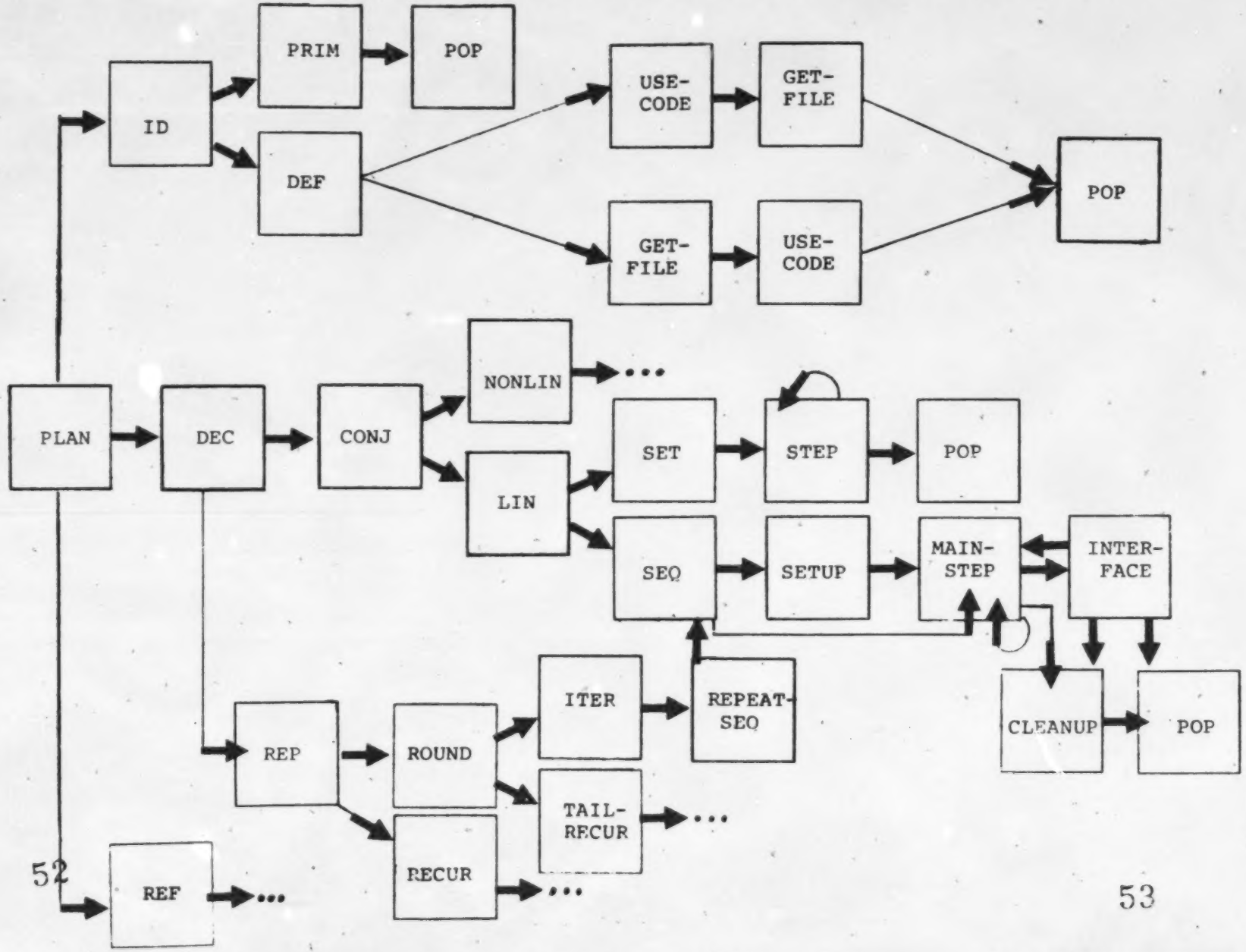
The ultimate version of SPADE ought to include a module for providing intelligent planning advice and filling in low level details of partially specified solutions. However, a context free grammar, being inherently non-deterministic, would not suffice as the basis for a machine problem solver. Solving problems by generating all possible derivations and then testing for a solution would hardly be practical.

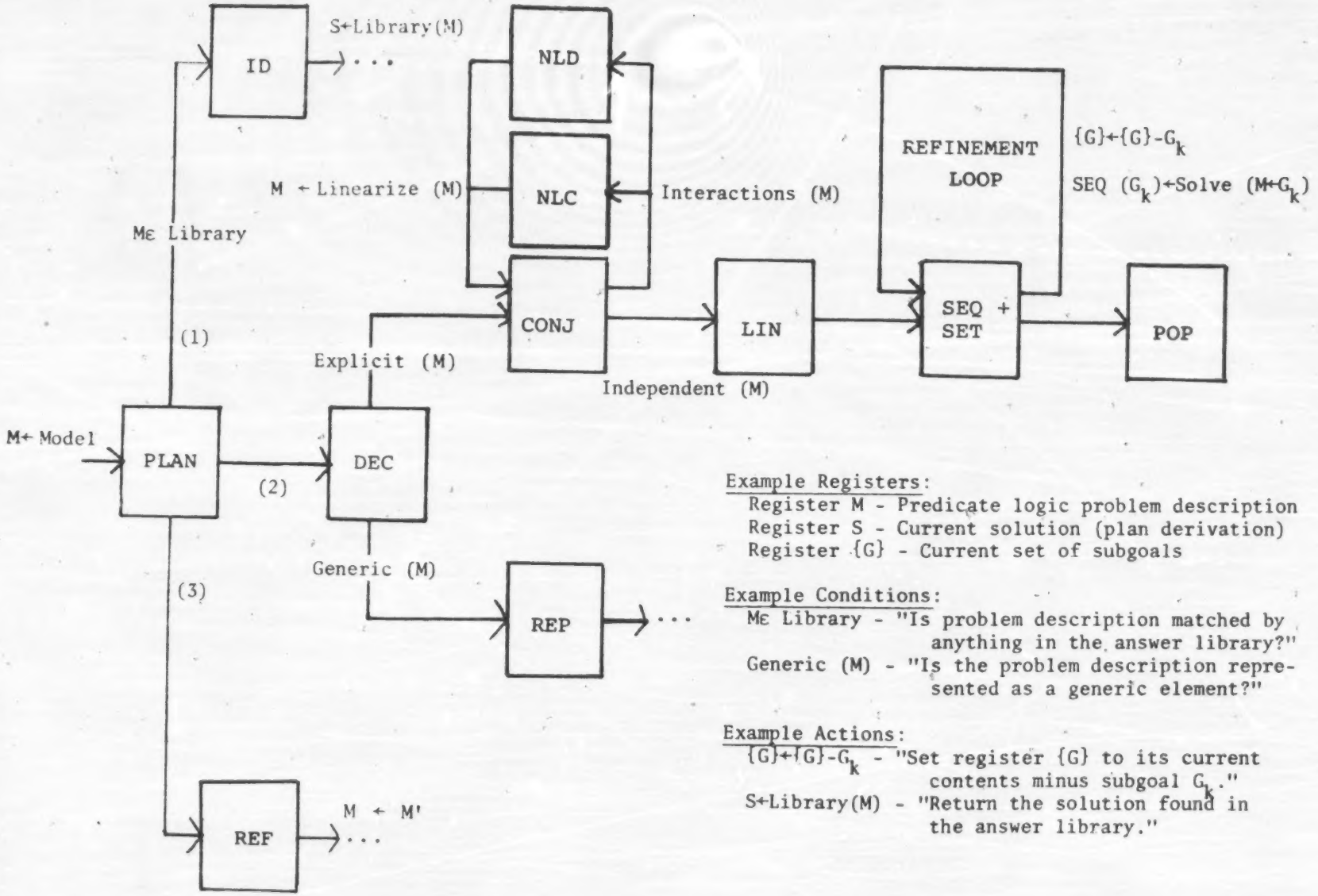
There is also a theoretical deficiency. There ought to be a facility for *skill acquisition*: for summarizing previous semantic or pragmatic planning errors to prevent their recurrence on similar problems in the future. Such a capability was exhibited by Sussman's [1973] HACKER program for example. But our context free grammar has no way of representing repair knowledge in such a way that semantic or pragmatic bugs are not repeated.

Both of these deficiencies can be addressed by moving from the context free grammar representation for planning knowledge to an augmented transition network [Woods 1970]. Augmented transition networks generalize the context free grammar representation. To see the way the ATN serves as a natural generalization of the grammar, first examine figure 22. Here we have an equivalent representation for the G2 planning grammar as a (non-augmented) recursive transition network. The *augmented* transition network provides several generalizations: (1) registers can be provided to store the values of variables; (2) predicates can be associated with arcs to control the order of transition; and (3) actions can be associated with arcs to build structures during transitions. These generalizations were introduced in computational linguistics to overcome limitations of the CFG representation that parallel those that we have met in the problem solving realm. Figure 23 is the planning ATN based on G2. Some (not all) of the registers, conditions, and actions (for storing and manipulating information about the current sub-problem) are shown. Notice how greater efficiency can be achieved via techniques such as collapsing states -- moving some information from the topological configuration to the registers (e.g., the CONJUNCTION and SEQ+SET nodes). Figure 24 shows how arc predicates can be used to select the appropriate plan type on the basis of features of the problem description. This approach (called *PATN*, for *Planning ATN*) is developed at length in [Goldstein & Miller 1976b]. Here our goal is only to show how repair skill could be acquired by SPADE/RAID by representing planning knowledge in an ATN.

Consider again the SQUARE-WITHIN-TRIANGLE problem discussed in the RAID section. Recall that the underlying cause of the bug was treating the SQUARE and

FIGURE 22 - (NON AUGMENTED) RECURSIVE TRANSITION NETWORK FOR G2





Example Registers:
- Register M - Predicate logic problem description
- Register S - Current solution (plan derivation)
- Register {G} - Current set of subgoals

Example Conditions:
- Mε Library - "Is problem description matched by anything in the answer library?"
- Generic (M) - "Is the problem description represented as a generic element?"

Example Actions:
- {G}←{G}-G_k - "Set register {G} to its current contents minus subgoal G_k."
- S←Library(M) - "Return the solution found in the answer library."

FIGURE 23 - AN AUGMENTED TRANSITION NETWORK FOR PLANNING

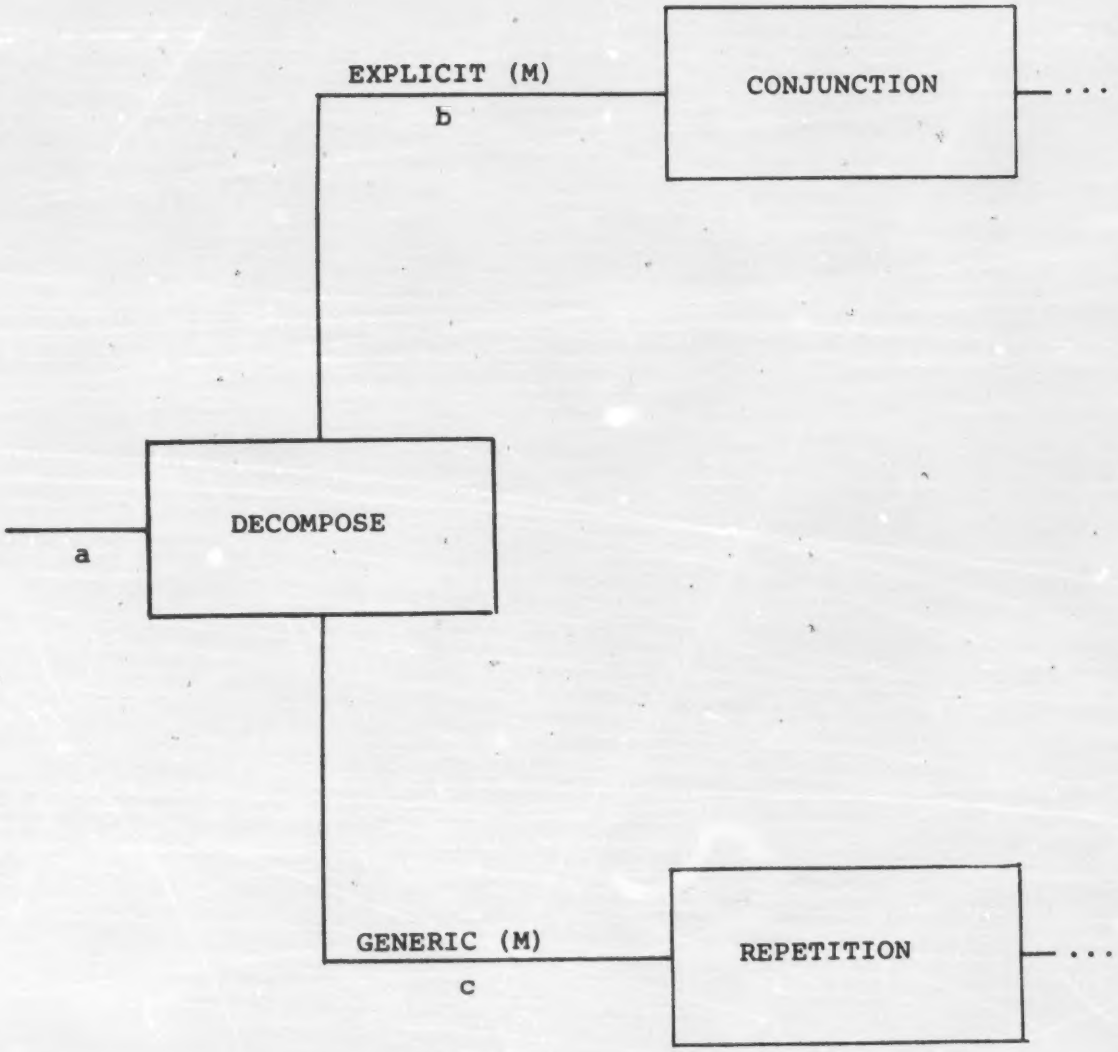


FIGURE 24

PATN'S DECOMPOSE NODE

TRIANGLE subgoals as if they were independent. In fact, a second order constraint on their sizes was imposed by the INSIDE restriction. Future occurrences of this error could be prevented by adding a condition that tests for the existence of the INSIDE predicate to the arc constraint that governs the selection of nonlinear plans.

Specifically, this is done as follows: figure 25 shows that part of PATN that corresponds to the rule,

P6: CONJUNCTION -> LINEAR | NONLINEAR.

The default arc ordering causes the LINEAR plan to be attempted first. The NONLINEAR transition is allowed only if the NLC or NLD predicates recognize the problem as containing a nonlinearity. Here, if INSIDE is present, the NLD loop is taken and the problem description modified to make the interaction explicit. A size predicate is added to the description of the parts. Thus, a new arc constraint, NLD-INSIDE, serves to prevent this particular pragmatic planning error from happening again.

6.2. Applications

These ideas lend themselves to a variety of applications. We consider three: automatic programming, automatic protocol analysis, and structured programming.

As semantic and pragmatic capabilities are added to SPADE (reflected by the increasing role of PATN in providing advice), the user would be consulted on progressively fewer planning decisions. The ultimate extension in this direction is of course for SPADE to request no guidance at all from the user. The user would supply the problem description; SPADE would provide the solution procedure. One novel aspect of this approach to automatic programming is methodological: the SPADE series of systems provides an implementation strategy based on *incremental simulation* [Woods & Makhoul 1973].

Automatic programming is an extension of SPADE in a direction in which the user is pushed toward the higher level planning decisions, whereas the system performs more of the lower level choices. Exploration in the opposite direction is also possible; in the extreme this amounts to *protocol analysis*. Suppose that the problem solving of a SPADE user is running far ahead of the system: the user may wish to type in code directly, rather than laboriously detailing the intermediate steps of the plan. The system's job then becomes linking the low level event into a higher level planning structure. If every event typed by the user were at this code level, SPADE would superficially be serving as a conventional editing environment. The difference would lie in the assistance

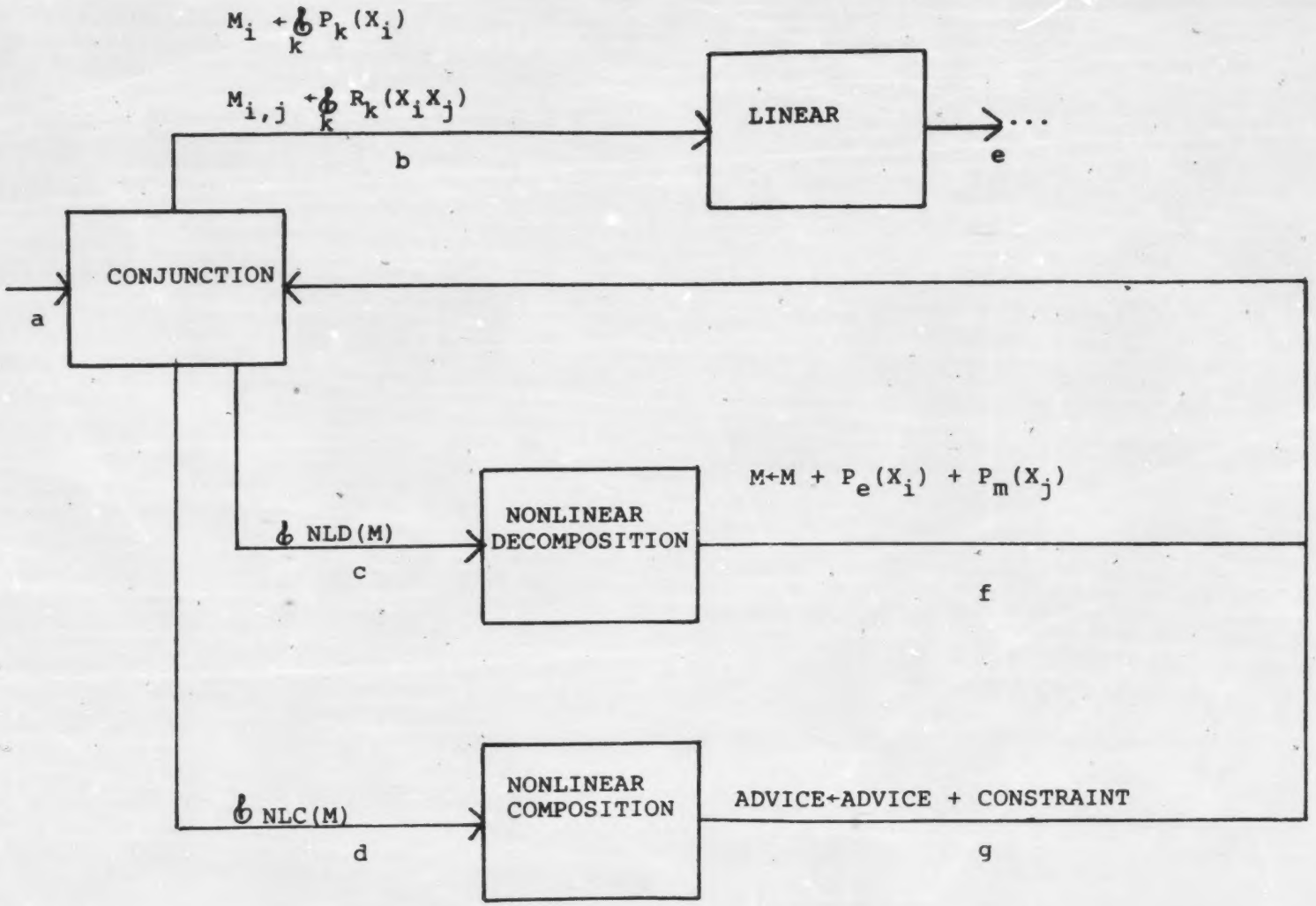


FIGURE 25

PATN'S CONJUNCTION NODE

possible during subsequent debugging. Ideally, SPADE would have a module (which we call *PAZATN*, for *Protocol AnalZer based on an ATN*) for inferring the user's *plan* -- and would therefore be able to support our deeper notion of debugging even when the plan is only implicit. Figure 26 illustrates how a hypothetical version of the SPADE system, augmented by PAZATN, could significantly reduce the amount of interaction required to articulate the planning knowledge *as well as the code* for use by the system. [Miller & Goldstein 1976d] takes a more careful look at the difficulties which the PAZATN module must face, and presents a preliminary design.

A final application is to prescribe improved programming methodology. The entire enterprise embodies Dijkstra's philosophy of programming in a structured fashion. Moreover, it represents a more detailed study of planning and debugging techniques than has previously been attempted. It indicates how interactive editors can strongly encourage coherently structured *articulate* planning. The underlying theory provides an analysis of the nature and origin of bugs, suggesting which sorts of bugs can be avoided by improved design, and which arise from justifiable heuristic choices. The occurrence of such uncertain choices however, can be recorded, leading to bookkeeping and diagnostic capabilities such as those planned for RAID. Better debugging advice -- going beyond caveats for potential difficulties -- must await the incorporation of PATN (and to some extent PAZATN) into SPADE.

This report has presented a unified theory of planning and debugging based on a linguistic analogy. The design of an interactive programming environment has also been described. The objectives for this programming environment, SPADE, are that it serve, not only as a practical application of the theory, but also as an experimental crucible for testing claims of the theory.

We expect that experimentation with SPADE will yield the following kinds of information: (a) AI evidence regarding the heuristic adequacy of the planning taxonomy and grammar; (b) psychological evidence regarding the utility of the grammatical formalism as a modeling tool, for characterizing varying skill levels, in terms of which subsets of the grammar are used successfully and unsuccessfully; (c) computer science evidence regarding the efficacy of alternative documentation standards and design methodologies; and (d) pedagogical evidence regarding the value for a learner of programming in this type of articulate environment.

Figure 26. A Scenario Illustrating SPADE Augmented by PAZATN

1a. We are now solving for a WISHINGWELL. Your pending subgoals are ROOF, POLE, WELL, and the interfaces. What now?
1b. >SQUARE

Here the user types in an event at level of the actual code. The proper inference is that the user is solving for WELL using an IDENTIFICATION plan. SQUARE is presumably in the user-defined answer library.

2a. OK. I am assuming that WELL has been solved by IDENTIFYing it with the previously solved SQUARE procedure. What now?
2b. >FORWARD 100

Again, the user has typed in a very low-level event which requires careful interpretation. There are a number of vector commands which might be expected synthetically. For example, this could be the interface between the WELL and the POLE. However, if the SQUARE used for the WELL is of size 100, then FORWARD 100 would be too large. Also, a preparatory rotation would have been needed. The vector might also be a side of the TRIANGLE for the ROOF. However, if TRIANGLE is already in the answer library, an identification would be expected, not a new solution. Probably, this vector accomplishes the next main step in PATN's default solution order: POLE. However, PAZATN can employ a demon to postpone a final commitment until further evidence arrives.

3a. *OK. What now?*
3b. *>TRIANGLE*

4a. *OK. I guess ROOF has been solved by IDENTIFICATION with the existing TRIANGLE. So the FORWARD 100 must be the POLE. Your only remaining subgoals are the interfaces. Which interface would you like to solve?*

.
.
.

It is worth noticing how few user type-ins have been required in this dialogue -- fewer than even in conventional code -- yet the solution for the WISHINGWELL is almost complete. Moreover, the system has inferred not only the code, but a rather thorough description of the user's plan as well. This economy of interaction would be achievable by the combination of SPADE, PATN, and PAZATN enabling the user to focus on the few critical planning choices that more or less force the remainder of the solution.

7. Notes

1. While there is some overlap, our objectives for SPADE differ in this respect from the objectives of those working to construct *programming apprentices* [Teitelman 1970,1974; Hewitt & Smith 1975; Rich & Shrobe 1975]. It is too early, however, for a detailed comparison of either goals or methods.

2. The virtues of the Logo graphics world [Papert 1971a,b; 1973] are: (a) graphics is an environment in which multiple problem descriptions are possible, ranging from Euclidean geometry to Cartesian geometry; (b) the possible programs range over a wide spectrum of complexity; and (c) there is extensive documentation on human performance in this area [G. Goldstein 1973; Okumura 1973].

3. These task domains are natural candidates for testing the generality of the theory. The blocks world is a benchmark AI domain which provides a yardstick against which to measure the progress of our approach. The set theory world has the virtues of both intrinsic interest and straightforward semantics. The creation of programs embodying concrete realizations of set theoretic constructs is a standard programming task. Similar remarks are appropriate for the domain of programming an elementary calculator, such as to perform routine statistical analyses.

4. In [Goldstein & Miller 1976a] we presented a scenario for a programming tutor called Sherlock. One extension of the SPADE system presented here is toward such *mixed-initiative* AI based personal learning environments. At the same time, the SPADE style of interaction suggests a more structured alternative approach. Whether the additional structure is desirable for some (or most) students is an empirical question to be addressed in future research.

5. In [Miller & Goldstein 1976b] we presented a different version (G1) of the planning taxonomy and grammar, in the context of parsing a student protocol. Our reasons for abandoning that version in favor of the current one should be discussed. The earlier taxonomy was based on examining the directions from which a planner could obtain guidance: looking upward to general principles, downward to domain specific heuristics, forward to anticipated needs, and backwards to previously solved problems. The current taxonomy derives from examining the logistic description of the current problem. The former taxonomy emphasized the roles of experimentation and uses of past problems; whereas the current one treats these as details, some of which (such as experimentation) remain to be addressed. It remains true that decomposition techniques can vary along dimensions of domain specificity and generality. However, in some cases we found the earlier taxonomy to be problematic. As we began to incorporate semantic and pragmatic constraints on the grammar (see [Goldstein & Miller 1976b]), it became

increasingly difficult to maintain a formal distinction between certain examples of domain dependent and evolutionary plans. There is of course a trade-off in assigning knowledge to the syntactic rules, as opposed to assigning it to semantic or pragmatic constraints on their application. In order to justify a claim that the current version of the taxonomy is more parsimonious than the previous one, we would need to carefully identify the corpus of data. While we do, in fact, believe that the current version is more elegant, the grounds for this belief remain intuitive. In subsequent research we intend to employ the SPADE system as an experimental vehicle for contrasting alternative planning taxonomies and their corresponding grammars.

6. The statement that there is a core set of planning techniques common to all domains is justified by examining the formal basis for the taxonomy. On the assumption that problem descriptions are represented as predicate calculus statements, it is clear that solution can proceed by: (1) identifying the statement as one for which a solution procedure is known to exist; (2) decomposing the problem into subproblems on the basis of the top level logistic operator; or (3) reformulating the problem description such as by theorem proving techniques. That is, domain independence of the core set of planning techniques holds, on *a priori* grounds, to the same extent that problems in the domain are describable using predicate calculus problem descriptions.

Of course this argument depends on the efficacy of the first order predicate calculus as a problem description language. While we are not prepared to argue for this here, it is clear that the calculus certainly has had some success in the past (e.g. in mathematics) and hence is an obvious candidate. Its frequently observed deficiencies, such as non-directed inferencing, are discussed in [Goldstein & Miller 1976b], where we define a procedural problem solver organized around logical operators. It is also important to recognize that we are not arguing for uniform (e.g., *resolution*-based) theorem prover style programming techniques.

Moreover, extensions of the predicate calculus, such as higher-order calculi, do not obviate the need for basic problem solving techniques for dealing with conjunction, disjunction, negation, and quantification.

7. This view of planning is a simplification. It asserts that the problem is analyzed in a top down fashion. Of course, the problem solver can engage in exploration and experimentation; or can identify a subgoal without having a clear understanding of the overall plan. The dynamics of exploration are not formalized by this grammar.

8. Our use of a context free grammar for problem solving closely resembles D. Rumelhart's [1975] work on story grammars. It should be interesting to see to what extent our respective theories, designed to account for superficially very different phenomena, continue to develop in parallel. Would it be useful, for instance, to define a set of *summarization rules* (such as those

employed by Rumelhart) to describe the planning process? One possible set of plan summarization rules would focus on the SOLVE nodes, suppressing printout for nodes of other types. Conceivably, this could be useful in highlighting the subprocedure organization.

9. The rules of the grammar are written using the following syntax:

disjunction:	"a | b" is read as, "a or b";
ordered conjunction:	"a + b" is read as, "a and b", where the order is significant;
unordered conjunction:	"a & b" is read as, "a and b", where the order is insignificant;
optionality:	"[a]" is read as, "a is optional";
iteration:	"<a>*" is read as, "a repeated 1 or more times";
lexical category:	a lower case English phrase enclosed in quotation marks (e.g., "number") describes a lexical item which is not further expanded in the grammar.

10. The & operator is used, because the GET and USE can occur in any order as long as they both precede execution of the procedure being defined.

11. While the Mycroft system designed by Goldstein was potentially capable of semantic annotation, it lacked a clear formalization of the range of possible planning choices a program designer could make, and a description of possible errors in terms of these design decisions. The grammar we present here is intended to address these limitations.

12. The interactions presented here are hypothetical dialogues with a system which has not been implemented. Although a crude preliminary implementation (SPADE-00) has recently begun, it is currently lacking several essential features.

One deficiency of SPADE-00 is that it has not been interfaced with LLOGO [Goldstein et. al. 1974]; hence it is not possible to actually execute the resulting programs. Another deficiency is that the RAID features described in a later section have not yet been coded.

The purpose of presenting hypothetical dialogues, rather than actual transcripts, is to enable the reader to focus on the content, without being

sidetracked by details concerning the inadequacies of the implementation. Readers who have access to the laboratory's timesharing system are nonetheless invited to experiment with our trial versions of SPADE as follows. After logging in, type :SPADE<cr>. :NSPADE will generally be a newer, highly experimental version. :OSPADE will be an older version, in case of disastrous malfunctioning by :SPADE.

SPADE-00 simplifies the interaction by employing a "menu" or "multiple choice" style:

```
WHAT WOULD YOU LIKE TO DO?
=A -- IDENTIFY
=B -- DECOMPOSE
=C -- REFORMULATE
>=a
```

Certain operations, such as the LATER capability, are implemented as special "escape commands," in order to reduce ambiguity and simplify parsing. For example:

```
>later
I DON'T UNDERSTAND:  LATER.
>@later
POLE POSTPONED.
```

Once started, the system is self-documenting, and is gradually becoming friendlier to use. Suggestions and bug messages may be sent via the system mailer to SPADE@MIT-AI.

13. The question arises as to whether the bug taxonomy is exhaustive when circumlocutions and slips of the tongue are also included. In a trivial sense, the answer is "yes" because the latter class is open-ended by definition. In a deeper sense, the answer *may* also be "yes" in that no bugs need ever be assigned to this category which violate our intuitive requirement that the underlying plan not be affected. This is a hypothesis which we tentatively accept but cannot prove.

14. To avoid possible confusion, it should be stressed that our bug classification does not correspond to the usual terminology of programming lore. While there is a slight analogy, it may be misleading. That is, "syntactic planning bugs" does *not* refer to the syntax of the *programming language*; it refers to the hierarchical structure of the *process of constructing* programs. Similar remarks are in order for semantic and pragmatic planning bugs. For brevity, we may use the shorter phrases, e.g., "syntactic bug," to refer to a syntactic planning bug. For the most part, we are not concerned here with syntax errors (or "semantic errors") in the usual sense.

15. A natural objection is that this particular bug could be eliminated if the computing environment were modified so as to automatically load appropriate files when needed. We completely agree. Indeed, it illustrates the point that the grammar illuminates the design of improved computing environments. It in no way alters the observation that, given any particular computing environment, certain syntactic constraints on the structure of programming plans must be adhered to, nor that violation of these constraints constitutes one type of error.

16. Of course, the issue arises as to whether the human problem solver is simply forgetting part of a known rule, or is unaware of the rule in the proper form. This leads to a set of difficult problems in protocol analysis surrounding the hypothesizing of the grammar underlying a given individual's problem solving. This topic is pursued in [Miller & Goldstein 1976b,d].

17. The occurrence of two consecutive calls to a given primitive is odd when a single invocation, perhaps with altered input, will suffice. In Logo, two adjacent PENUP commands, or two adjacent FORWARD instructions would be considered rational form violations.

18. This wishingwell program employs what Goldstein [1974] termed an "insertion plan." The TREE shown here is inefficient in that it achieves state transparency via retracing the TRUNK. There is also a tradeoff in the WELL between modularity and efficiency. The use of the insertion plan to avoid retracing on the top side of the WELL results in less modular code. These points are noted only to avoid misunderstanding -- they have no bearing on the thrust of the example.

19. Viewing debugging from the vantage point of this taxonomy sheds some light on the issue of the *pedagogical value* of various kinds of bugs. Our current understanding of the first three (and to some extent the fourth) categories of bugs suggests that encounters with such bugs may be instructive in teaching planning as well as debugging. However, "slips of the tongue" at best provide some exercise in bug localization. Hence, a "forgiving" system that minimizes the penalties for such low level bugs is probably pedagogically sound. The best example of this philosophy is the Interlisp *DWIM (Do What I Mean)* facility [Teitelman 1970,1974]. (By contrast, our effort to make more of the *plan* explicit might be called *SWIM*, i.e., *Say What I Mean*!)

20. If we define a context free grammar for debugging, then this error in debugging technique can be classified. For example, if the rule of the debugging grammar for fixing slips of the tongue is:

REPAIR -> [UNDO] + REDO

where undoing is optional since it is not always required, then the error of failure to undo (due to forgetting or to confusion regarding the existence of side effects) is semantic.

An alternative view of debugging would be to characterize planning as a context free grammar, while debugging is described as a *transformational* component that maps derivation trees to derivation trees. This would be theoretically elegant, and this possibility deserves further study. However, resolution of this issue goes beyond the current paper.

21. Later we briefly introduce a module we are designing called PAZATN which would help to alleviate this difficulty. PAZATN would be capable of *parsing* programming protocols, inferring -- from a combination of synthetic expectations and analytic evidence -- which plans had been used.

22. A fundamental hypothesis of the Logo project is that children learn by doing and *thinking about what they do*. One of our purposes in implementing the SPADE editor is to explore this hypothesis by experimenting with the relative merits of SPADE versus the traditional Logo programming environment. In SPADE, the student is required to be articulate. Whether this helps students to master planning and debugging concepts more quickly -- or hinders them -- remains to be seen. Our conjecture is that despite the extra interaction demanded, students will find the need to be articulate about their problem solving a significant help in learning, as measured by an ability to solve harder problems more quickly.

8. References

Dahl, Ole-Johan, Edsger Dijkstra and C.A.R. Hoare, *Structured Programming*, London, Academic Press, 1972.

Goldstein, Gerrianne, *Logo Classes Commentary*, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Logo Working Paper 5, February 1973.

Goldstein, Ira P., "Understanding Simple Picture Programs," in *Artificial Intelligence*, Vol. 6, No. 3, 1975; and Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Technical Report 294, September 1974.

Goldstein, Ira. P., Henry Lieberman, Harry Bochner, and Mark L. Miller, *LLOGO: An Implementation of LOGO in LISP*, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Memo 307 (Logo Memo 11), June 27, 1974.

Goldstein, Ira P., and Mark L. Miller, *AI Based Personal Learning Environments: Directions For Long Term Research*, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Memo 384 (Logo Memo 31), December 1976a.

Goldstein, Ira P., and Mark L. Miller, *Structured Planning and Debugging: A Linguistic Theory of Design*, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Memo 387 (Logo Memo 34), December 1976b.

Hewitt, Carl, and Brian Smith, *Towards a Programming Apprentice*, IEEE Transactions on Software Engineering, vol. SE-1, no. 1, March 1975.

Kaplan, Ronald M., "A General Syntactic Processor," in Randall Rustin (ed.), *Natural Language Processing*, Courant Computer Science Symposium 8 (December 20-21, 1971), New York, Algorithmics Press, 1973, pp. 193-241.

Kay, Martin, "The MIND System," in Randall Rustin (ed.), *Natural Language Processing*, Courant Computer Science Symposium 8 (December 20-21, 1971), New York, Algorithmics Press, 1973, pp. 155-188.

Miller, George A., Eugene Galanter and Karl H. Pribram, *Plans and the Structure of Behavior*, New York, Holt, Rinehart, and Winston, 1960.

Miller, Mark L., *Cognitive and Pedagogical Considerations for a Tutorial Logo Monitor: An Investigation Into the Evolution of Procedural Knowledge* (S.M. Thesis), Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, February 1976.

Miller, Mark L., and Ira P. Goldstein, *Overview of a Linguistic Theory of Design*, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Memo 383 (Logo Memo 30), December 1976a.

Miller, Mark L., and Ira P. Goldstein, *Parsing Protocols Using Problem Solving Grammars*, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Memo 385 (Logo Memo 32), December 1976b.

Miller, Mark L., and Ira P. Goldstein, *PAZATN: A Linguistic Approach To Automatic Analysis of Elementary Programming Protocols*, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Memo 388 (Logo Memo 35), December 1976d.

Okumura, K., *Logo Classes Commentary* Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Logo Working Paper 6, February 1973.

Papert, Seymour A., *Teaching Children to be Mathematicians Versus Teaching About Mathematics*, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Memo 249, 1971a.

Papert, Seymour A., *Teaching Children Thinking*, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Memo 247 (Logo Memo 2), 1971b.

Papert, Seymour A., *Uses of Technology to Enhance Education*, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Memo 298 (Logo Memo 8), June 1973.

Polya, George, *How to Solve It*, New York, Doubleday Anchor Books, 1957.

Polya, George, *Mathematical Discovery* (Volume 1), New York, John Wiley and Sons, 1962.

Polya, George, *Mathematical Discovery* (Volume 2), New York, John Wiley and Sons, 1965.

Polya, George, *Mathematics and Plausible Reasoning* (Volume 1), New Jersey, Princeton University Press, 1967.

Polya, George, *Mathematics and Plausible Reasoning* (Volume 2), New Jersey, Princeton University Press, 1968.

Rich, Charles, and Howard E. Shrobe, *Understanding LISP Programs: Towards a Programmer's Apprentice* (Master's Thesis), Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, August 1975.

Rumelhart, David, "Notes on a Schema for Stories," in D. Bobrow and A. Collins, *Representation and Understanding: Studies in Cognitive Science*, New York, Academic Press, 1975.

Sussman, Gerald Jay, *A Computational Model of Skill Acquisition*, New York, American Elsevier, 1975; and Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Technical Report 297, 1973.

Teitelman, Warren, "Toward a Programming Laboratory," in J.N. Buxton and B. Randell (eds.) *Software Engineering Techniques* (Report on a Conference sponsored by the N.A.T.O. Science Committee, Rome, Italy, October 1969), April 1970, pp. 137-149.

Teitelman, Warren, *INTERLISP Reference Manual*, Cambridge, Bolt, Beranek and Newman, October 1974.

Woods, William A., "Transition Network Grammars for Natural Language Analysis," *Communications of the ACM*, Volume 13, Number 10, October 1970, pp. 591-606.

Woods, W.A., and J. Makhoul, "Mechanical Inference Problems in Continuous Speech Understanding," BBN Report 2565, Bolt Beranek and Newman Inc., Cambridge, Mass., August 1973.